



University
of Glasgow

Gdura, Youssef Omran (2012) *A new parallelisation technique for heterogeneous CPUs.*

PhD thesis

<http://theses.gla.ac.uk/3406/>

Copyright and moral rights for this thesis are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

A New Parallelisation Technique for Heterogeneous CPUs

by
Youssef Omran Gdura

A Thesis presented for the degree of

Doctor of Philosophy

to the School of Computing Science,

College of Science and Engineering,
University of Glasgow



UNIVERSITY
of
GLASGOW

May 2012

Copyright © Youssef Omran Gdura, 2012.

Abstract

Parallelization has moved in recent years into the mainstream compilers, and the demand for parallelizing tools that can do a better job of automatic parallelization is higher than ever. During the last decade considerable attention has been focused on developing programming tools that support both explicit and implicit parallelism to keep up with the power of the new multiple core technology. Yet the success to develop automatic parallelising compilers has been limited mainly due to the complexity of the analytic process required to exploit available parallelism and manage other parallelisation measures such as data partitioning, alignment and synchronization.

This dissertation investigates developing a programming tool that automatically parallelises large data structures on a heterogeneous architecture and whether a high-level programming language compiler can use this tool to exploit implicit parallelism and make use of the performance potential of the modern multicore technology. The work involved the development of a fully automatic parallelisation tool, called VSM, that completely hides the underlying details of general purpose heterogeneous architectures. The VSM implementation provides direct and simple access for users to parallelise array operations on the Cell's accelerators without the need for any annotations or process directives. This work also involved the extension of the Glasgow Vector Pascal compiler to work with the VSM implementation as a one compiler system. The developed compiler system, which is called VP-Cell, takes a single source code and parallelises array expressions automatically.

Several experiments were conducted using Vector Pascal benchmarks to show the validity of the VSM approach. The VP-Cell system achieved significant runtime performance on one accelerator as compared to the master processor's performance and near-linear speedups over code runs on the Cell's accelerators. Though VSM was mainly designed for developing parallelising compilers it also showed a considerable performance by running C code over the Cell's accelerators.

Declaration

This thesis presented a work that was carried at the University of Glasgow under the supervision of Dr. William Cockshott and Dr. John O'Donnell , School of Computing Science, during the period between March 2007 to January 2012. I declare that the work is entirely my own work and it has not been previously submitted for any other degree or qualification in any university.

Youssef Omran Gdura

Glasgow, May 2012

Acknowledgments

I would like to express my grateful acknowledgment to my supervisors Dr. William Paul Cockshott for his guidance, enthusiasm and assistance during the course of this PhD and to my second supervisor Dr. John O'Donnell for his support. I also wish to express my sincere thanks to the School of Computing Science, University of Glasgow for creating an environment that has been fabulous for research and fun. Also this work would not have been possible without the financial support from the Ministry of Education in Libya, I am grateful to them for this opportunity.

To my external and internal examiners, Prof Sven-Bodo Scholz and Dr Wim Vanderbauwhede, for their interest in this work and for taking the time to study this thesis extensively.

I would like also to take this opportunity to express my sincere appreciation to all the staff at University of Glasgow for their support and valuable assistance during the Libyan crisis in 2011.

I dedicate this work for the soul of my father and mother, who passed away while I was doing my Master degree and to the soul of my oldest brother and oldest sister, who passed away while I am doing my PhD.

Special thanks to my beloved wife, Nadia, for her patience and providing me comfort and to the five parallel accelerators in my life my daughters, Raian, Raihan and Jeanan and my sons, Omran and Abdalrahman, for giving me their love, support and all the joy. I am most appreciative of my brothers and sisters for their continuing support, as without their supports and wishes, I couldn't achieve what I have achieved today. Above all, I thank Allah almighty for his unlimited blessings.

Youssef

Glasgow, May 2012

Contents

Abstract	i
List of Figures	v
List of Tables	viii
List of Publications	ix
1 Introduction	1
1.1 Thesis Statement	3
1.2 Motivations	4
1.3 Objectives	5
1.4 Contributions	6
1.5 Outline	7
2 Background	10
2.1 Overview of Parallel Processors	10
2.2 Parallel Memory Architectures	12
2.3 Parallel Computing	13
2.4 Parallel Programming Paradigms	22
2.5 Virtual Machines	33
2.6 Genetic Algorithms	34
3 The Cell Processor & Vector Pascal	37
3.1 Introduction to PowerPC Architectures	37
3.2 The Cell Broadband Engine Processor	47
3.3 Glasgow Vector Pascal	60
3.4 Assembly Language Directives	67
4 Related work	68
4.1 Compute Unified Device Architecture (CUDA)	68
4.2 Open Computing Language (Open CL)	71
4.3 Programming the Cell BE Architecture	75

5	Virtual SIMD Machine	96
5.1	Introduction	96
5.2	Virtual SIMD Registers	98
5.3	Virtual SIMD Instructions	100
5.4	VSM Messaging Protocol	102
5.5	PPE Interpreter	104
5.6	SPE Interpreter	109
5.7	Using VSM	123
5.8	Experimental Results	125
6	Host Compiler Development	135
6.1	PowerPC Machine Description	136
6.2	Machine-dependent Routines	140
6.3	Assembly Macros	142
6.4	Stack Frame Operations	144
6.5	Compiler Building Process	145
6.6	PowerPC Compiler Extension	146
6.7	Building the VP-Cell Compiler System	152
6.8	Coding	154
7	Code Generator Optimiser	155
7.1	Introduction	155
7.2	Previous Work	156
7.3	Permutation Technique	157
7.4	Why Instructions Ordering is a Problem	158
7.5	Genetic Algorithm Approach to the Problem	161
7.6	Key Design Aspects	162
7.7	Implementation	167
7.8	Experimental Results	171
8	Evaluating VP-Cell Compiler	179
8.1	Machine Configuration	179
8.2	Testing Developed Tools	180
8.3	Performance Tuning	181
8.4	Experimental Results	182
9	Conclusion and Future Work	206
9.1	Contribution	207
9.2	Future Work	208

List of Figures

2.1	Arrays Alignment	17
2.2	Sample of APL's Character Set	22
2.3	ZPL Directions Operators	25
2.4	Defining Arrays in SaC	28
2.5	SaC WITH loop	29
3.1	Pascal Code and CISC Assembly Instruction	38
3.2	Pascal Code and RISC Assembly Instruction	38
3.3	PowerPC Instruction Format	39
3.4	PowerPC ABI registers conventions	40
3.5	The Cell BE Schematic Diagram	49
3.6	SPE Hardware Diagram	50
3.7	SPE_CONTROL_AREA Structure	56
3.8	SPE Hardware Diagram	60
3.9	Pascal Code Segment	62
3.10	A Simple VP Pure function	65
4.1	OpenCL Functions	74
4.2	Scalar C Function	74
4.3	Parallel OpenCL Kernel	74
4.4	OpenMP Program Execution	79
4.5	Sieve Function Prototype	81
4.6	Sieve Block	81
4.7	SieveC++ Implementation	82
4.8	Sieve C++ Code	83
4.9	Offload Function Prototype	85
4.10	Offload C++ Code	86
4.11	C++ Class	93
4.12	C++ Code Segment Using TBB libraries	93
5.1	Splitting a VSM Register on 4 SPEs	98
5.2	Message Formats	103

List of Figures

5.3	Broadcasting PPE Messages to SPES	105
5.4	SPE Thread Creation	106
5.5	Launching SPE Thread Using POSIX Threads	106
5.6	Barrier Synchronisation Routine	107
5.7	Store Virtual SIMD Instruction	109
5.8	Load Virtual SIMD Instruction	110
5.9	SPE Interpreter Structure	111
5.10	Messaging Pulling Code Segment	112
5.11	Loading Unaligned Data to SPE's Local Memory	114
5.12	Splitting an SPE Block into 3 DMAs for Storing Process	116
5.13	Synchronise Shared-Block	118
5.14	Storing Middle Block	119
5.15	Acknowledgment of an Operation Completion	119
5.16	Add Operation	121
5.17	Replicate Operation	122
5.18	Pattern Of Reduction Operation	122
5.19	Vector Add Reduction Operation	124
5.20	Building VSM Object Files	124
5.21	A C Program Uses VSM as API	126
5.22	C Simulator	127
5.23	(a) shows the latency per thread Launching while (b) approximates the percentages of time spent in the activities involved in setting up a single SPE thread	128
5.24	The Latency of Sending Messages to SPEs	130
5.25	The average latency time for processing a 32-bit word (single precision floating-point) using a virtual register of size 4 KB per SPE.	132
5.26	The time for processing block of data on 1,2 and 4 SPEs. Each operation was measured using a block of 4096 single precision floating-point values, a virtual register of size 4 KB per SPE and run 10^6 times	133
5.27	(a) The percentage of time spent on the main activities out of the total latency time of each operation (b) The percentage of time spent on processing aligned data as compared to unaligned data.	133
6.1	Building the code generator and compiling a Pascal program	153
7.1	Branch Instructions Patterns in ILCG Using Gas Assembly	158
7.2	Assemble Code	159
7.3	Optimal Assembly Code	160
7.4	One-point Crossover	164

List of Figures

7.5	The performance of a genetic algorithm on optimizing a PowerPC instruction set ordering.	174
7.6	Register-to-Register Operation	175
7.7	Register-to-Immediate Operation	176
7.8	The results gained by optimizing previously manually optimised code generator for the Pentium using a genetic algorithm. The average fitness values reflect the performances of the code generators on the three selected applications.	177
7.9	Normalized performance gained by optimizing instruction set ordering of PowerPC and Pentium 4 architecture using a genetic algorithm.	177
8.1	Simple Program in VP to Test a BLAS Kernel	183
8.2	Generated Code for Expression $x := y + z$	185
8.3	(a) Performance PPE vs. SPE	186
8.4	Performance and Scalability of SPEs	187
8.5	C code to compute the dot product of two vectors in Sequential on the PPE and in parallel using the Cell's SPEs.	188
8.6	Performance of C code on the PPE and the SPEs by Using VSM as an API	190
8.7	SPE Scalability On BLAS Kernels Using C Code.	191
8.8	N-body Problem Pseudocode	192
8.9	Performance of One SPE vs. the PPE (N-body Problem)	194
8.10	Speedups of Vector Pascal on the Cell's processors (N-body Problem)	195
8.11	The SPEs Performance	196
8.12	SPE Scalability	197
8.13	Vector Pascal and C Performance	199
8.14	Performance of SPEs versus PPE on Blurring Program	204
8.15	SPEs Scalability on Blurring 4096x4096 Image Using Different VSM Register Sizes	205

List of Tables

2.1	Flynn's taxonomy	14
5.1	VSM Opcodes	101
5.2	The average latency of the basic thread management operations on the Cell accelerators	127
5.3	The average time of processing a block of 4096 a single precision floating-point (32-bit) elements using a virtual register of size 4 KB per SPE. Each operation was run 10^6 times	131
6.1	Emulating Intel ENTER instruction on PowerPC	144
7.1	Genetic Algorithm Improvements on the PowerPC. The first set of applications were training programs and the last two programs were not in the training set.	176
8.1	Machine Configuration	179
8.2	Basic Linear Algebra Kernels	184
8.3	Performance of Vector Pascal vs. C.	199
8.4	Performance of the PPE and SPEs Using Different VSM Register Sizes	204

List of Publications

- 1. Gdura, Y. and Cockshott, P., A Virtual SIMD Machine Approach for Abstracting Heterogeneous Multi-core Processors, Journal of Computing (GSTF), vol. 1, number 4, pp. 143-148, 2012.**
- 2. Gdura, Y. and Cockshott, P., A Compiler Extension for Parallelizing Arrays Automatically on the Cell Heterogeneous Processor, in Proceedings of the 16th International Workshop on Compilers for Parallel Computing (CPC2012), Italy January 2012.**
- 3. Cockshott, P., Gdura, Y. and Keir, P., Two Alternative Implementations of Automatic Parallelisation, in Proceedings of the 16th International Workshop on Compilers for Parallel Computing (CPC 2012), January 2012.**
- 4. Cockshott, P. and Gdura, Y., Code Generator Optimizer Using Genetics Algorithm Techniques, 1st Asia-Pacific Programming Languages and Compilers Workshop (APPLC), 2012. (Accepted)**
- 5. Cockshott, P., Gdura, Y. and Keir, P., Array Languages and the N-body problem, Concurrency and Computation Journal: Practice and Experience, CPE-12-0022. (Submitted)**

1 Introduction

The notion of concurrency and parallel computing have existed from the earliest development of supercomputers, and since then the promises of parallelism have fascinated many researchers [1, 2, 3, 4, 5]. Parallel computing is a fundamental research topic in computer science and grew as a topic of interest in the mid 1980's upon the introduction of massively parallel processors and networks of computers [1, 3, 6, 7, 8, 9, 10, 11]. The interest in this area has also been stirred up in the last two decades by the advent of the Single Instruction Multiple Data (SIMD) technology in the 1990's and later multi-core platforms in the mainstream industry such as multi-core general purpose architectures (CPUs) and Graphics Processing Units (GPUs) [12, 7, 13, 14, 15]. These modern architectures offer high performance hardware with reasonable cost that made them widely used in many application areas, such as image processing, graphics, multimedia, modeling and scientific computation [16, 17, 7, 18, 9]. This widespread industry adoption of multi-core architectures has a significant influence on mainstream software and applications development as they require proper, simple to use and up to date tools for developing parallel and concurrent programs.

This chapter starts with a short introduction to key issues in parallel computing and the targeted processor and language. It then presents the thesis statement, motivations, contributions and concludes with a summary of what will be covered in each chapter.

The key issues in parallel programming are: identifying available parallelism, partitioning data, managing data transfers between cores, handling required communication among cores, synchronisation between cores and performing any required data alignments [19, 20, 6, 21, 22, 23]. This turned out to be a very complicated task [3, 24, 25, 10, 8, 9], and software researchers, in the last two decades, have put additional effort on developing various programming paradigms that simplify the parallelisation process. Yet success in developing simple and fully implicit parallel models has been limited due to the complex analysis that compilers are required to do [26, 23, 17, 27]. In recent years, various tools such as programming languages, extended compilers, libraries and APIs have been developed to simplify parallel programming on multicore architectures [19, 18, 28, 29, 30, 31, 32, 33, 34, 35]. The simplicity of a given model is often measured by its capability to handle the parallelisation process and most importantly how to identify available parallelism.

The three common approaches to exploit parallelism in a given application are: Explicit, Semi-implicit and Implicit techniques. Fully explicit-based models require substantial details, such as identifying parallelism, message passing, data movement, alignment and synchronisation, to be provided manually by programmers. In contrast, semi-implicit parallel programming models liberate programmers from several parallelisation tasks such as thread creation, communication, data transfers, synchronisation and alignment, but they still depend on programmers to identify parallel regions of code using annotations, pre-processor directives and constructs [25, 9, 10]. Developers during recent years have had good success in applying this technique in various programming models for parallel architectures. OpenMP is the most commonly used semi-implicit programming model nowadays for shared memory multi-core architectures [30, 8, 36]. Hybrid Multi-core Parallel Programming, called OpenHMPP, is also directive-based model that has been recently introduced for developing parallel programs on CPUs and GPUs [37]. Software developers have also introduced a number of compiler extensions such as SieveC++ and library routines, such as OffloadC++ and Intel Threading Building Block (TBB) [38, 18, 20].

There are also programming languages that have been built to support parallelism without relying upon annotations and directives mechanisms. High Performance FORTRAN (HPF) and Matlab [19, 39], for example, offer constructs for expressing parallelism in an implicit manner. The Z programming language (ZPL), Vector Pascal (VP) languages and Fortran are another programming paradigm that uses arrays abstraction to implement a data parallel programming [40, 41, 42]. Functional programming languages, such as Single Assignment C (SaC), Glasgow Parallel Haskell (GpH) and its extension Data Parallel Haskell (DPH), are mutation-free programming models that also provide support for concurrency mechanisms [43, 34, 44]. Chapel, Fortress and NESL are also parallel programming languages, but they have been developed for supercomputers such as the Thinking Machines CM5 and the Cray C90 [45, 46, 33, 47]. Virtual Machines such as CellVM and Hera-JVM, have been developed as experimental models to manage multithreaded applications on parallel architecture in particular the Cell processor [5, 48, 49]. Most of the programming models that have been mentioned so far are for programming general purpose processors. CUDA and OpenCL are new parallel languages that have recently emerged for programming general propose application on specialised processors such as GPUs and digital signal processing (DSP) [50, 51, 52, 53].

Nevertheless, explicit and semi-implicit based models are often unfavorable for parallelising existing sequential applications due to the required alterations and adjustments and sometimes do not guarantee to improve the performance. The third technique for exploiting parallelism is known as an implicit or automatic technique. Implicit-based models require no programmers interference as the whole parallelisation process is left to their compilers. Yet it is an open question whether it is it possible to develop a parallelising

compiler that discovers all the parallelism available in an application. The short answer probably is auto-parallelising compilers often do not succeed in delivering full parallelisation because finding parallelism usually requires a sophisticated compiler analysis and code mapping [3, 23, 6, 24, 25, 11].

Heterogeneous architectures make it even harder for compilers to generate efficient parallel code than with homogeneous machines. Heterogeneous multi-core architectures have different types of processing cores, and each type is designed to support and carry out a different set of functions such as the Cell heterogeneous architecture. The Cell processor, which is targeted in this work, is a single multi-core chip processor that consists of a general Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs) [54, 55, 56]. The Cell processor potentially offers high levels of parallelism, but it is not easy to program due to its heterogeneity of CPUs, memory structures and instruction sets. The PPE is responsible for overall control of the processors while the SPEs are mainly designed for computation. Cell is a distributed-shared memory architecture which has main memory space on the PPE and private Local Storage (LS) on each SPE. An SPE's LS can be accessed directly by the SPEs and by the PPE but only through DMA controllers in a non-coherent mode [56, 57, 54]. The other heterogeneity of the Cell appears on the instruction level as the PPE and SPE's have different instruction sets. This feature presents a major challenge for developing Cell applications because it requires writing source code for each core type and the use of two compilers. The Cell architecture showed performance potential specially when using good heterogenically adapted code [56].

However, this work focuses on developing a programming tool that can aid high level programming languages compiles to automatically parallelise arrays operations, and therefore Glasgow Vector Pascal (VP) was chosen for this project. It is an extension of the standard Pascal programming language, and it supports SIMD instruction set extensions and data parallel operations [58, 42, 7]. Vector Pascal is a suitable choice of language because it supports array syntax and array operations. Operating on entire arrays rather than loops with explicit indexing provides opportunities for compilers to generate parallel code. Yet, the most important feature in this language is that its front-end compiler already supports SIMD technology and flexible degrees of parallelism, and hence its code generator can be easily switched to produce look-alike SIMD code that operates on large registers.

1.1 Thesis Statement

Heterogeneous multi-core architecture use is still limited due to the difficulties associated with programming and managing their heterogeneity. The heterogeneous technology has

introduced new challenges for compiler developers on how to make parallel programming easier for ordinary programmers and to make the most of the existing parallel platforms. I assert that a new virtual machine approach can be employed to reduce the burden of developing applications for modern general purpose heterogeneous processors while enabling automatic parallelisation of computations on large data sets. The work shall be demonstrated by designing and implementing a Virtual SIMD Machine (VSM) model that hides the intricate details of the Cell heterogeneous architecture completely and allows parallelising array operations on its accelerators. The work shall also investigate whether an array programming compiler, such as Glasgow Vector Pascal (VP) compiler, can be extended to use the model to exploit data parallelism implicitly and attain sufficient performance. The VSM and the extended VP compiler are expected to work together as one compiler system that takes a single source code and examines the code for conventional array expressions. If an array expression includes arrays of adequate size for parallelisation, the compiler then delegates the evaluation of the array expression to the Cell accelerators, otherwise the evaluation is performed on the master processor.

The ambitions are to reduce the complexity associated with the fully automatic parallelisation approach by focusing only on array expressions, and also to ease the task of developing programming parallel applications by concentrating on algorithms rather than on parallelisation issues such as communication, partitioning, alignment, and synchronisation.

1.2 Motivations

It's very obvious that manually developing parallel programs is time consuming, error prone and costly, besides parallelising existing applications manually usually requires significant transformation of code [25, 9, 5, 34, 38, 47, 2, 59]. In the last two decades, more effort has been put into simplifying the difficult task of explicitly managing parallelism on multi-core architectures, and consequently the quality of the parallelising tools has been improved after the advent of semi-implicit parallel programming models [30, 60, 20, 50, 52]. The semi-implicit based models have been considered effective parallelisation tools to save time and effort, yet they still require programmers' involvement to explicitly identify or express parallel components and to verify if it is worthily parallelising or not. The verification process is essential in order to guarantee performance improvement otherwise parallelisation may result in performance degradation. The identification of available parallelism is also considerably important. Therefore, an attractive programming approach is to have a parallelising compiler that implicitly spots parallel

computations in sequential applications, but this would be also a very complicated task for a compiler attempting to discover all the parallelism available in an application.

In the light of these considerations, this work suggested developing a programming tool that hides all the underlying details of the Cell processor and assists existing compilers to focus on parallelising arrays expressions on the Cell accelerators. Arrays are a good target data structure for several reasons. First arrays are a good candidate for data parallelism because operations on arrays are inherently parallel operations and can easily be applied to various sections of an array in different ways. Secondly, evaluating one array expression at a time minimises the memory requirement as compared to evaluating a compound statement or using block offloading techniques. This proposed technique focuses on evaluating an individual array expression which relatively requires a small space and suits architectures with limited amount of local storage well. Arrays also provide opportunities for compilers to directly generate parallel code without the need for a complex analysis process to search for available parallelism or the need for programmers' interference to identify parallel regions of code.

1.3 Objectives

- To provide a fully implicit programming model for parallelising array operations and supporting scalable parallelisation on heterogeneous architectures such as the Cell architecture.
- To completely hide the underlying details of heterogeneous architectures where the accelerates behave as coprocessors.
- To introduce a framework that can be used as an abstract model to shorten the time for developing parallelising compilers for heterogeneous architectures.
- To simplify the complexity of the analytic process that a compiler is required to do to identify available parallelism.
- To reduce the burden involved in developing parallel programs for general purpose heterogeneous processors such as the Cell processor.
- To enable programmers to focus on using the proper parallel algorithms rather than on parallelising code.
- To reduce the execution time of the automatically parallelised programs as compared to the execution time of the sequential ones.

1.4 Contributions

The following contributions have been made by this work:

- The VSM approach for abstraction heterogeneous multi-core processors:
 - Presenting a new parallelisation approach which imitates the SIMD concept to increase the work a single instruction performs.
 - Developing a fully implicit programming model for parallelising array operations and supporting scalable parallelization
 - Demonstrating the possibility to hide completely all the underlying details of a heterogeneous multi-core architecture such as the Cell processor.
 - Introducing a framework that can be used by existing compilers to parallelise arrays expression on heterogeneous multi-core architectures.
 - Developing a VSM interface as an abstract model to shorten the time for developing parallelising compilers
 - Presenting a messaging protocol that can be used to exchange information between different processing core types.
 - Developing alignment and synchronisation algorithms to handle alignment constraints, maintain data consistency and to avoid race conditions while accessing main memory by a machine's accelerators.
 - Hiding completely all low-level details of memories management such as data alignments, transferring messages between different cores and data transfers between the core memory and all devices.
 - Presenting a fully implicit model that can be used as API by high-level programming languages to perform array operations in parallel on heterogeneous architecture using a single-source code compiler.
- The PowerPC Back-end Compiler
 - Developing a back-end compiler to port Glasgow VP to the Cell's master processor.

- Developing a new approach to optimise the code-generator of a compiler. The novelty of this approach is to apply a genetic algorithm to automatically optimise machine instructions ordering. This optimiser was not part of the objectives, but it was developed along the way to optimise the VP back-end compiler for the Cell's master.
- Extending the PowerPC back-end compiler to work with the VSM model as a single compiler system.

1.5 Outline

Chapter 2 presents background information on parallel computing. It starts with an overview of parallel processors and parallel memory architectures. It then talks about different levels of parallelism and the SIMD technology. After that, it looks at parallel programming techniques and existing array parallel programming models such as the A programming language (APL), the Z programming language (ZPL), Single Assignment C language (SaC) and Fortran language.

Chapter 3 is devoted to delineate the targeted processor and programming language. It begins with background information on the PowerPC architectures and the machine-dependent features such as stack frames and functions calling conventions. It then describes the Cell heterogeneous cores, the main component of the Cell accelerators, and the communication mechanisms that the Cell supports. This chapter shall also introduce the main library functions that are provided in the manufacturer's Software Development Kit (SDK). The attempt is to discuss only the SDK library functions that shall be used in this work such as the functions for exchanging messages and thread creation. After that, it introduces the Glasgow Vector Pascal programming language and its features. The discussion includes more details on how to build VP back end compilers and the required tools. It discusses also some issues that are closely related to this work such as array boundary checking and vectorising array operations.

Chapter 4 introduces a number of programming paradigms that have been recently developed for programming general purpose applications on heterogeneous shared-memory architectures. It first sheds some light on the two parallel programming models, CUDA and OpenCL, which have been developed lately to make use of GPUs for general purpose computing. After that, it presents the parallel programming models that have been introduced recently for programming the Cell processor. Two of these models have been

introduced as commercial tools such as OpenMP and Offload C++. OpenMP has already been in use on other architectures. It shall introduce also two other models, CellVM and Hera-JVM which have been designed specifically for the Cell processor.

Chapter 5 describes a Virtual SIMD Machine (VSM) interface. It starts with a description of the VSM registers. It then describes the type and format of virtual machine instructions. It will also present samples of different instruction implementations. It next explains the developed messaging protocol from both the design and implementation retrospectives. After that, it discusses in detail the VSM's two co-operative interpreters. The discussion will introduce two new algorithms that were designed and implemented to handle the alignment and synchronisation of the VSM load and store instructions.

Chapter 6 discusses the conventional VP back end compiler and the extended version for the Cell processors. It first describes the PowerPC machine specification of the conventional compiler and gives an example to illustrate how the instructions order can effect generated code. It then describes the machine-dependent routines and the two stack frame operations: ENTER and LEAVE. After that, it describes and discusses the extended version of the compiler which allows it to parallelise array expressions automatically. The discussion includes virtual SIMD registers set, new VSM instructions set and any adjustment on the machine-dependent routines which have just been mentions. The chapter concludes with the description of how to use the extended compiler and the VSM model as one compiler system.

Chapter 7 discusses in detail a code generators optimiser. The optimiser is based on genetic algorithm techniques to automatically optimise machine instruction ordering. It starts with an introduction of the problem and a description of the basic algorithm. It then looks at key design aspects and the implementation. After that, it presents the experimental results that show the genetic algorithm's improvement and the quality of automatically constructed code generators.

Chapter 8 presents the experimental results which showed that the VP-Cell system achieved significant runtime performance on the Cell's accelerators and also a considerable performance by using the VSM model for parallelising C code over the Cell's accelerators. This chapter first discusses the observed performance of the VSM model on basic array operations using BLAS micro-benchmarks written in Vector Pascal. It then provides detailed analysis on the performance of the VP-Cell compiler based on real-world

benchmarks. It will also present experimental results obtained using VSM model as an application programming interface for parallelising a number of C linear algebra kernels.

Chapter 9 presents the conclusion and explores the opportunities for the future that build upon the work presented in this dissertation.

2 Background

This chapter starts with an overview of parallel processor and parallel memory architectures and then discusses in brief the principles of parallel programming. It then looks at a number of parallel programming paradigms. The discussion focuses most directly on array programming languages since the purposed VSM model was initially designed to back up array programming compilers in parallelising array expressions. After that, it looks in a brief at genetic algorithms and how they were previously used in optimising generated code.

2.1 Overview of Parallel Processors

The interest in parallel hardware dates back to the beginning of the 1960s when Slotnick proposed the design of the first vector processing machine [61], yet his design was implemented one decade later by the University of Illinois to produce the ILLIAC IV system [61]. This machine was organized in four quadrants. Each quadrant had its own control unit and contained 64 processors, and each processor had 6 registers, 2K words of local memory and operated on 64-bit words [61]. The quadrants were designed as individual units to be partially used for applications that did not need the computation power of the entire machine, and eventually only one quadrant was built. The ILLIAC IV was one of the first machines that supported Single Instruction Multiple Data (SIMD) technology that basically offers the means to perform the same operation on multiple data concurrently [3].

Processors that offer support to SIMD technology are considered as a class of parallel architectures [7, 62]. By the time the ILLIAC IV was ready to be used publicly, it was taken by the introduction of the first Cray computer (Cray-1) in 1976 [62]. Cray-1 was designed with pipeline vector arithmetic units to operate on large data sets, and its CPU sustained performance reached 138 MFLOPS [3]. A second SIMD computer was introduced in 1979; it was the ICL Distributed Array Processor (DAP) that had 64x64 processor elements each processing a single bit at the time. Unlike the ILLIAC IV, the ICL DAP was a commercial product, and one of the users was the University of Edinburgh which used

it for several science applications [3]. In 1982, the Cray X-MP supercomputer was released. It was a shared memory parallel vector architecture with two central processing units. It had a theoretical peak performance of 400 MFLOPS [63]. During that period two Massively Parallel Processor (MPP) supercomputers were introduced [64, 3]. Goodyear, which was produced in 1983, was one of the first MPP systems. The second system was the Connection Machine (CM-1), and it was also introduced in the early 1980s by the Thinking Machine Corporation [65]. CM-1 consisted of 64K bit-serial processors, and its design also supported SIMD instruction sets. In the late 1980s, Cray Research started the production of the Cray Y-MP supercomputer series [66]. The first machine in this series was known as Y-MP Model D. This Model was designed to have 2,4 or 8 vector processors with multiple computation units each. The Y-MP production line continued until the 1990s. The vector processing and massively parallel processor supercomputers are used for solving complex problems, but they are very costly and require special skills to be programmed. These boundaries besides the demand for midrange computer machines led to the development of minicomputers which progressively developed to microcomputers or single-user machines.

The 12-bit Programmed Data Processor (PDP) computer was one of the first minicomputers. It was produced by Digital Equipment Corporation in 1965 with practical capabilities and reasonable cost for small groups. Minicomputers as stand alone machines were not designed as parallel computers, yet two or more minicomputers were commonly used to build a system that worked as a parallel computer. As minicomputers developed in the 1970s and 80s, a new small class of computing machines emerged in the 1980s with the advent of microcomputers or single-user machines (aka Personal Computers PC's) [3, 4]. Single-user computers with a single core processor have become very popular since the 1990s as their performance consistently advanced and their costs were relatively affordable. The performance of single core processors kept advancing during the 80s and 90s by enhancing their resources, such as clock speed, memory size and storage devices, and adopting the SIMD technology [4, 67, 7, 68]. The power of conventional single core machines, however, had reached in the late 1990s some physical limitations like speed, heat dissipation and power consumption. According to Moore's Law, which describes the long-time expectation of hardware development, computers' performances were anticipated to double every 18 months as a result of the number of transistors that can be placed in a given area. Yet in recent years the single-core architectures clock frequency and other resources have reached a plateau [68, 69]. These limitations as well as the demand for high performance mainstream machines to keep up with power needed for various applications, which are used on a daily basis, shifted designers toward multiple core solutions: Special purpose computing machines that integrate specialized processing elements such as Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and most com-

monly Graphic Processor Units (GPUs) and general purpose computing machines with multiple core versions of conventional CPUs [70, 71, 59, 72].

The fast growth of the game and graphics market along with the physical limitations of the single core technology led to the advent of GPUs in the mainstream industry. GPUs are highly parallel architecture that are traditionally designed for computer graphics and dedicated to calculating floating point operations. The first GPU with hardware-acceleration was released in 1999 by Nvidia, and it was called GeForce 256 [70]. It was the first fully 3D accelerator. Modern GPUs have also been improved by allowing software developers to use the GPUs for non-graphics computations, but they are relatively expensive and hard to program. These burdens plus the necessity for high performance machines for general purpose computing applications have led also to the introduction of general purpose multi-core architecture.

However, the mainstream industry has been focusing recently on the development of processors with a relatively small number of cores that theoretically are expected to offer a satisfactory performance for general purpose computing. Most modern multicore chips are symmetric platforms, such as AMD Phenom II with 6 cores and Intel Nehalem with 4 cores, and both were released in 2008 [9, 73]. The alternative is heterogeneous multi-core chips, such as 9-core IBM Cell processor which was produced in 2007 [54, 56, 74]. Heterogeneous multicore architectures often have specialized cores that are designed for specific functionality, and they are expected to provide better performance as compared to symmetric platforms [68, 9]. The VSM model was implemented to parallelise array operations on the Cell heterogeneous processor.

2.2 Parallel Memory Architectures

Parallel memory architectures can be distributed-memory, shared-memory or a mix of both. A distributed-memory system, also called a private-memory system, consists of a number of computers' processors connected via a general interconnection network, and each processor has its own private memory and can operate independently [5, 4, 75]. This type of structure does not require memory coherence protocols because updating a location on one processor's memory will not affect the data on the memories of the other processors. One of the advantages of a distributed-memory system is that it makes building a large-scale network of heterogeneous machines with high performance computing possible. The other advantage is that each processor has very fast access to its local memory. This feature can also be found in Symmetric Multiprocessing (SMP) platforms. The third advantage is that distributed-memory systems are scalable in terms of memory size

because as the number of processors increases the total size of memory increases [76]. On the contrary, distributed-memory platforms are not appropriate for applications that are based on global data structures and require programmers to explicitly define how and when data is transferred between processors. Distributed memory systems also require a message-passing protocol to communicate between processors [31, 8], and one of the widely used parallel programming model for managing distributed-memory systems is Message Passing Interface (MPI).

A shared-memory system refers to a global address space that can be simultaneously accessed by multiple processing elements [8]. This form of architecture is often an Uniform Memory Access (UMA) system that makes a global physical memory equally accessible to all processing elements [76, 75]. The communication between processors in a shared-memory system is maintained by memory coherence protocols which notify the involved processors of changes to shared memory resources. This technology offers a fast media of communication between multiple processors, and it is relatively easy to program because there is only one source of data. The most popular parallel programming model for shared-memory platforms is OpenMP. I shall elaborate more on OpenMP in the next chapter.

Comparing the two parallel memory systems, software problems are more complex in distributed-memory systems than in shared-memory systems, while hardware problems are easier in distributed-memory systems [75]. The two systems can be, however, combined together in what is known as Distributed Shared Memory (DSM) systems. A DSM system generally involves a number of nodes connected by an interconnection media, and each node, in addition to its own local memory, has access to a shared memory [75]. DSM systems could be a set of clusters connected by interconnection network or could be built on a smaller scale such as the Cell processor. On the Cell processor, each accelerator has its own local memory and shares with other accelerators the system memory.

2.3 Parallel Computing

Parallel computing is a fundamental research topic in computer science and grew as a topic of interest in the mid 1980s upon the introduction of massively parallel processors and networks of computers [1, 3, 6, 77, 78, 8, 9, 10, 11]. The interest in this area has also been stirred up in the last two decades by the advent of the Single Instruction Multiple Data (SIMD) technology in the 1990s and later by merging multi-core platforms, such as multi-core general purpose architectures (CPUs) and special purpose platforms like Graphics Processing Units (GPUs), in the mainstream industry [12, 7, 14]. These

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Table 2.1: Flynn’s taxonomy

modern technologies offer high performance platforms with reasonable cost that made them widely used in many application areas, such as image processing, graphics, multimedia, modeling and scientific computation. However, this widespread industry adoption of multi-core architectures has a significant influence in mainstream software and applications development and has introduced new challenges for software developers to provide proper, simply-used and up to date tools for developing parallel and concurrent programs. The main challenges in parallel programming are: managing data transfers between cores, handling the required communication among cores, synchronization between cores, performing any required data alignments and exploring parallelism available in applications [19, 20, 6, 21, 17, 79, 38, 22, 23]. The following sections shall discuss two of the key problems of parallel programming: the SIMD technology and alignment issues and then look at the key issues to identify parallelism such as levels of parallelism, types of parallelism and parallelisation techniques

2.3.1 SIMD technology

In 1962, Michael Flynn proposed what is known as Flynn’s taxonomy for categorizing computer machines [80]. Flynn’s categorization, was based on data streams that can be manipulated by instructions or control units, and it was divided into four classes as shown in Table 2.1. The discussion in this section shall focus only on the SIMD class or technology since the VSM parallelisation model emulates the conventional SIMD concept.

SIMD technology includes suitable techniques for applications that are inherently parallelisable such as media processing, 3D graphics and games. This technology is a software model that extends a machine instruction set to allow operations on multiple data concurrently and consequently offers performance improvement [7, 81]. This technology has been in use since 1970 by supercomputers such as CM-1 and ICL DAP [82, 65]. The CM-1 and CM-2 machines were capable to carrying out 64K 1-bit arithmetic operations simultaneously [65]. The ICL DAP was also a SIMD architecture that had the capability to perform 4096 arithmetic operations at a time [82].

This technology has been through several developments especially after it has being used in mainstream computers. Sun Microsystems adopted this technology in the mid 1990s by introducing SIMD integer instructions in its UltraSPARC I machine. Around the same

time, Intel introduced the MMX instruction set with its fifth generation of Pentium architectures [83]. MMX was integer instructions that operate on existing floating point registers. Yet sharing one set of registers caused a problem as processing units can not work on both floating point data and SIMD integer data at the same time [7, 14]. In 1998 Advance Micro Devices (AMD) also released its 3DNow! instruction set to support SIMD technology on the AMD K6-2 architectures [84]. The first implementation of the 3DNow! supported only SIMD floating-point operations [84]. One year later, Intel introduced with the Pentium III processor another instruction set called Streaming SIMD Extension (SSE). The SSE extension solved the problem that MMX had by using independent registers (called XMM). Further advancement was provided by the SSE extensions series; SSE2, SSE3 and SSE4, to support computation in parallel on 128-bit integers and floating point [14]. Apple, IBM and Motorola also developed their SIMD extensions (AltiVec) to the Power architectures [85, 14, 7]. AltiVec is a 128-bit extension and has its own register file. AltiVec architectures do not use scalar integer and floating point units, instead they use a 128-bit vector processing unit [85]. Nowadays most machines provide 128-bit SIMD instruction sets, yet Intel introduced in 2011 its Advanced Vector Extensions (AVX) on the Sand Bridge processor family [86]. AVX is a 256 bit set extension to Intel's previous SSE [81, 86].

SIMD instructions allow processing multiple data items in a single operation [7]. The scale of performance that SIMD machines can provide is based on the length of machine registers and the size of the data type to be processed. For example, the level of parallelism that Intel SSE series and IBM AltiVec can support ranges between two and sixteen words at a time depending on data type. The number of words processed at a single step, however, can be doubled by using the Intel Advanced Vector Extensions (AVX), which was introduced in 2011. Intel AVX instruction set has been extended to operate on 256 bit instead of 128 bit [81]. The technology has been introduced as part of the Sand Bridge processor family [81]. These wider vectors improve performance as more data elements can be processed in a single operation.

A number of software tools, such as compilers and library functions, have been developed to make use of the parallelism offered by SIMD technology [12, 7, 14]. In 1997, Stanford University introduced a C parallelising compiler for the Sun UltraSPARC architecture using SIMD extensions [7, 87]. Intel also developed a C++ compiler that provides a set of built-in functions to support MMX and the SSE series. Codeplay's VectorC compiler was also designed to produce vector code for SIMD machines. The VP compiler, which was developed at Glasgow University, is also designed to support an SIMD instruction set such as MMX and 3DNow [58, 7, 35]. In this project, the SIMD technology was imitated using a virtual SIMD instructions that operate one large vector (virtual) registers.

2.3.1.1 The Data Alignment Problem

Data alignment is the way data is laid out and accessed, and the rules, which control it, are different from one machine to another. On an original machine instruction set, most architectures handle a load and store instruction to unaligned data internally. For example, on most RISC machines any attempt to access memory locations that are not aligned will generate an alignment fault [88, 89]. This fault is usually solved internally by the operating system using byte loads or stores to overcome the problem [88]. Other machines, like x86 architectures, did not require aligned memory access while processors, such as MIPS, have special unaligned memory access instructions [88].

However, in order to exploit the best performance of SIMD architectures, SIMD memory access instructions must be aligned because accessing unaligned data can considerably effect the efficiency of SIMD vectorisation [7, 90, 14]. Most modern architectures support SIMD extensions which require data to be aligned [7, 90, 14, 89]. For example, the Cell's SPEs instruction set is an SIMD extension that operate only a 16-bytes boundary, and if data is unaligned the compiler then must explicitly handle that by using, for example, rotate or shuffle operations to align and extract data [85]. Alignment problem became one of the challenges in using SIMD instructions sets as data in many applications is unaligned. For instance, unaligned dynamic memory accesses in the MediaBench and SPEC95fp benchmarks represent around 86% of the total memory access [90].

The alignment problem, in fact, is also one of the most important issues in the VSM design which meant to imitate the SIMD technology using bigger (virtual) registers. Because VSM is designed to parallelise arrays operations on multiple cores, this would highly lead to having the sub-array boundaries unaligned. To illustrate this point let us consider using the Cell's SPEs which require data to be aligned to 16-byte or 128-byte boundaries. Assume that the arrays A, B and C are 32-bit floating point arrays and all are aligned to a 128-byte boundary, and the size of A is 1024, B is 2048 and C is 4096 elements. Also, assume that data is divided equally when multiple SPEs are used. Now, let us start with the first expression shown in Figure 2.1. The VP compiler will deal with this expression as if it operates on arrays of 1024 elements. The decision is based on the size of the left hand side of the expression, and thus such expressions are valid as far as the R.H.S. arrays are equal or bigger than the one on the L.H.S of the assignment operator. Provided that, the compiler will add the first 1024 elements in B to the corresponding elements in C and store the results in array A. Evaluating this expression on 1, 2, 4 and 8 SPEs will comply with the SPE's alignment constraints and will give correct results. It is also expected to improve the SPEs performance because the operation operates on the same corresponding elements in the arrays, A, B and C, and the starting addresses of each sub-part of the three arrays on each SPE adhere to both cache line and hardware alignment rules even when the arrays

```
A:=B+C;  
A[1..1024]:=B[15..1038]+C[2111..3134]
```

Figure 2.1: Arrays Alignment

are partitioned on 2,4 and 8 SPEs. However, if a parallelising tool was strictly designed to divide data equally, the tool then can not parallelise such arrays on 3 or 6 SPEs.

The second expression, which is shown in Figure 2.1, is also a valid array expression in VP. It uses the same arrays A , B and C , but it operates on sub-ranges of the arrays. Though the starting address of arrays B and C are aligned, the first SPE will start from element 15 in B and element 2111 in C and these locations are not aligned. This raises two separate but related matters: Validation and Performance. Evaluating the second expression without handling the misaligned starting addresses of the sub-parts of B and C will generate an alignment error when attempting to load these sub-parts. This would be also a problem when attempting to store the results if array A started from a memory location that is not aligned instead of starting from the first element as it is in Figure 2.1. One solution to this alignment problem is to load additional bytes in temporary data buffers and then shift, rotate or copy the data to aligned buffers. The problem of storing unaligned data is even more complicated and challenging than loading additional bytes because it requires process synchronisation. However, fixing alignment problems must not generate alignment errors and most importantly must produce the expected results. This process often comes at the expense of the slower performance, yet one can sacrifice performance to get concurrency but not results. The alignment and synchronisation algorithms that the VSM model employed to solve these issues will be discussed in detail in Chapter 5.

2.3.2 Levels of Parallelism

Parallelism can be exploited at four different levels: bit-level parallelism, instruction-level parallelism, data-level parallelism and task-level parallelism.

2.3.2.1 Bit-Level Parallelism

A bit-level parallelisation is based on increasing a machine word size. The word size on early microprocessors was only 4-bit, and since then it has been doubling from 8-bit to 16-bit, then to 32-bit, and lately to 64-bit. The 64-bit architectures are now leading the market.

2.3.2.2 Instruction-Level Parallelism

The Instruction-level Parallelism (ILP) techniques are based on the number of instructions that can be executed at a time. Pipelining instructions execution is one of these techniques. Pipelining, in principle, implies dividing an instruction processing into several stages. A processor with an N-stage pipeline can have up to N instructions at different stages of completion. This technique became available in the 1960s after the introduction of machines with multi-stage instruction execution and has become the de facto standard technique in modern architectures.

Issuing more than one instruction at a time is another technique to exploit parallelism on instruction level. Processors with multiple execution units can issue different instructions to different units. For example, on the Cell processor each accelerator has two execution units that are known as Even and Odd units. The Even unit is designated for memory access and branch instructions, while the Odd unit handles computing instructions [56]. These two execution units allow the execution of two instructions at a time, for instance, by routing a “Load” instruction to the Even unit and at the same time issuing an “Add” instruction to the Odd unit [91]. There are also other techniques to exploit instruction-level parallelism including out-of-order execution and register renaming [92].

2.3.2.3 Task Parallelism

Task parallelism (aka function parallelism) basically focuses on splitting a big problem into sub-tasks. The sub-tasks can then be implemented as threads and executed across parallel processors. The two important issues about task parallelism are: load balancing and synchronization. Sub-tasks execution times often vary dynamically from one sub-task to another, and hence poor load balancing may degrade performance. Synchronization is also an issue of concern in task parallelism because improper synchronization may result in race conditions or additional overhead costs. These issues also hold for data parallelism.

2.3.2.4 Data Parallelism

This form of parallelism concentrates on splitting data into small blocks that can be processed in parallel. The history of data parallel programming began with the introduction of vector processors in the 1970s, and it has grown widely since the 1990s after a wide range of modern processors have instruction-set extensions for performance improvement specially in multimedia applications [7]. Data parallelism as compared to task parallelism appears to be promising since:

- Many applications are data-intensive in nature.
- Easier to spot data parallel operations than to find subsystem tasks specially in imperative and array programming languages.
- Simpler to assign blocks of data to individual processors than to split code into subprograms assuming homogeneity in operations.

In contrast, there are challenges associated with data parallel execution. The fundamental challenges are:

- Identifying data parallel regions
- Automating data partitioning
- Accessing shared data that may result in a race condition problem.
- Synchronizing common data access to avoid race condition
- Synchronizing operations to avoid data inconsistency

2.3.3 Identifying Parallelism

The simplicity of a given model is often measured by the capability of its compiler to handle the parallelisation tasks and most importantly how to exploit available parallelism. Identification of available parallelism is a significant step in the parallelisation process, and there are three possible approaches to identify parallel regions of code: explicit, semi-implicit or implicit techniques.

2.3.3.1 Explicit Parallelism

Explicit-based models, such as POSIX threads (Pthreads) and the Message Passing Programming (MPI) [93, 31], require substantial details, such as identifying parallelism, message passing, data movement, alignment and synchronization, to be provided manually by programmers. This approach, which is usually referred to as fully explicit parallelism, is time consuming, error prone, and costly. The explicit approach allows programmers to have the control on tuning a given code for better performance, but sometimes it makes the programmers' task difficult because the programmer have to specify the parallel parts and manage, and this could result in performance degradation if the selected parts were not worthily parallelisable. For these reasons, software researchers, in the last two decades,

have put additional effort on developing various techniques for managing parallelism automatically. Yet success in developing simple and implicit identification techniques has been limited due to the complex analysis that compilers are required to do in order to identify available parallelism in sequential applications [26, 23, 17, 27].

2.3.3.2 Semi-Implicit Parallelism

Most of the widely used parallel programming models nowadays are capable of handling parallelisation tasks like communication, data partitioning and synchronisation automatically, but they still require the interference of programmers to identify parallel regions in source code. This is why they are called semi-implicit models. These models depend on annotations, preprocessor directives and constructs to identify the parts of the source code to run in parallel leaving the parallelisation process, such as data movement and synchronization, to the compilers. Developers during recent years had good success in applying this technique in various models for programming parallel architectures. Some of these models, such as OpenMP and OffloadC++, depend on preprocessor directives and annotations to exploit available data parallelism [30, 60].

OpenMP is the most commonly used semi-implicit programming model nowadays for shared memory mutli-core architectures [30, 8, 36]. Other models focus on task parallelism such as Parallel Virtual Machines (PVM), CellVM, Hera-JVM and Intel Threading Building Blocks (TBB) [49, 48, 20, 5]. These models are also explicit-based tools that offer similar identification techniques which are initially designed to exploit parallelism at threads level. For example, CellVM and Hera-JVM depend on annotations to define sub-tasks in applications, while Intel TBB depends on a C++ template library to ease using standard threading paradigms such as POSIX threads [94]. CUDA and OpenCL are programming models that have recently merged in the parallel environment for programming specialised platforms. They have been developed to support general purpose computation on processors such as GPUs and digital signal processing (DSP) [50, 51, 52, 53]. Hybrid Multicore Parallel Programming, called OpenHMPP, is also a directive-based model that has been recently introduced for developing parallel programs on CPUs and GPUs [37]. There are also some models that rely on Libraries to exploit data parallelism explicitly. For example, CUDA Data Parallel Primitives (CUDPP), which runs on platforms that support CUDA, is a library of data-parallel algorithms that can be used for parallelising primitive operations such as Reduction and Sort operations [95].

Though semi-implicit parallel programming models liberate programmers from several parallelisation tasks such as thread creation, communication, data transfers, synchronization and alignment, they still depend on programmers to identify parallel regions of code

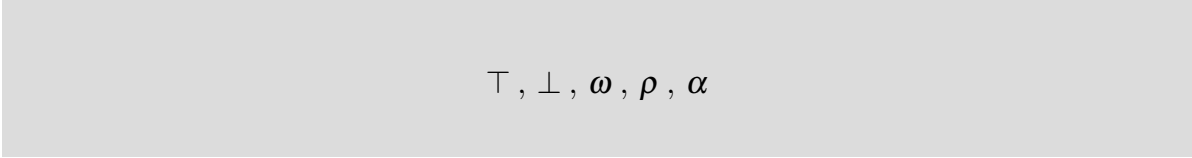
[25, 9, 10]. Explicit and semi-implicit based models are also unfavorable for parallelising existing sequential applications due to the required alterations and adjustments and do not guarantee to improve the performance. In view of these considerations, it will be practically productive if there is a tool that can automatically spot and parallelise available structures in sequential applications and frees ordinary programmers from managing parallelism explicitly [25, 9, 8, 79, 10].

2.3.3.3 Implicit Parallelism

The third approach for exploiting parallelism is known as an implicit or automatic technique. Implicit parallel programming models must have the capability to automatically manage parallelism starting from thread creation and identifying available parallelism, going through data partitioning, communication, alignment and ending with synchronisation. That is, this approach is expected to facilitate the development of parallel programs and to direct programmers to focus on solving problems in parallel fashion and not needing to think about the parallelisation process.

The implicit approach is a practical technique to extract parallelism already inherent in the structure of a programming language such as array and functional languages. High Performance Fortran (HPF) and Matlab [19, 39], for example, offer constructs for expressing parallelism in an implicit manner. ZPL, Fortran 90 and VP languages depend also on arrays abstraction to express data parallelism [41, 96, 97, 40, 42, 34]. Functional languages such as SaC and GpH are based on a computation model that also suits concurrency mechanisms [98, 43, 44]. Mutation-free programming models, like functional languages, are suitable for parallel programming at threads level because functions can be easily evaluated in parallel [99, 100, 101]. Chapel, Fortress and NESL are also parallel programming languages but developed for particular supercomputers. However, such programming languages, which have the basis to support parallelism, can depend on compiler technologies to implicitly identify opportunities for parallel code without the need for annotations and directive [6].

Though implicit-based parallel programming models are very convenient and practical for parallelising existing applications, the implicit techniques tend to be often very complicated and hard to implement because it requires a complex code analysis to determine the part of code to run in parallel [102, 103, 25]. Also, many of the implicit parallel languages are not commonly used because programmers often prefer to use languages that they are more familiar with. Actually, these considerations were what motivated us to think about developing a fully implicit programming model such as the VSM model which focuses only on parallelising arrays operations to narrowing the target code and to ease the task of existing compilers specially for well known programming languages.



$$\top, \perp, \omega, \rho, \alpha$$

Figure 2.2: Sample of APL's Character Set

2.4 Parallel Programming Paradigms

Programming paradigms that allow expression parallelism can be dated back to the introduction of the first imperative array language; A Programming Language (APL). Since then, there have been various programming paradigms introduced, such as functional and array programming languages, APIs models and virtual machines, for developing parallel programs. However, these paradigms have used different approaches to exploit and handle parallelism.

The discussion below focuses most directly on array programming languages since the proposed VSM model was initially designed to back up array programming compilers in parallelising array expressions. The following sections introduce the array programming languages: APL, ZPL and Fortran90. It then touches briefly on Single Assignment C (SaC) and virtual machines technology as the proposed VSM model is a virtual machine based design. The recently developed parallel programming models like OpenCL, OffloadC++ and OpenMP shall be discussed in the next chapter.

2.4.1 A programming language

The A Programming Language (APL) is one of the pioneer imperative languages that introduced the concept of arrays as data objects and has been the most influential of data-parallel languages. It is based on a mathematical notation invented by K. Iverson in 1957 [104]. APL in its first releases used an unusual character set and symbols rather than words [104]. Some of these symbols are shown in Figure 2.2.

APL was first used by the IBM System/360 computer in the early 1960s [105]. After that APL/1130 was introduced for the IBM 1130, and few years later used by the IBM 5100 desktop computer which consisted of a keyboard and a small CRT [106]. Then an advanced version of the language, called Sharp APL, was released in 1979 by a Canadian firm named I.P. Sharp Associated. The Sharp APL version added a number of language extensions such as shared variables, nested arrays, file system and packages for grouping objects. In the 1980s, IBM released the APL2 programming language for its mainframes such as CMS and TSO. APL2 included new features to improve the use of nested arrays,

and it was considered as a standard for the next APL interpreter developments [106]. In the 1990s, Iverson introduced an advanced version of APL called the J language. J uses an ASCII character set instead of the special symbols used in the previous releases of APL [107].

The APL language provided workspace, like in MATLAB, in which a user can define programs or operate on data without the need to write programs. For example, the following line,

$$V \leftarrow 1\ 2\ 3\ 4$$

assigns the values 1,2,3 and 4 to vector V , and the next line

$$+/V$$

performs a reduction add operation on vector V and displays 10 as a result.

APL also offers library routines for handling linear algebra operations which made it powerful in performing linear operations such as matrix multiplication. The APL language is available on Windows and UNIX platforms and offers interpreters and compilers for both platforms [106, 108, 26]. APLNext, Dyalog and MicroAPL are advanced interpreters that operate on Windows, Unix, and Linux. APL is often thought of as an interpretive language, yet there are also a number of APL working compilers. Most of the APL compilers are source-to-source compilers that translate source code to a low level language, such as C, and then use intermediate language compilers [26]. In the mid 1980s, Budd developed an APL compiler that worked under UNIX on the VAX-780 vector machine [108]. Budd's compiler is a parallel compiler based on implicit techniques to exploit parallelism in APL programs [108].

The language has been used in many fields and for different purposes such as mathematical, economic, accounting research, and simulation applications, and it has great influence on several programming languages such as J, K, Mathematica, MATLAB, SaC and Glasgow Vector Pascal [106, 108, 58, 109, 44]. It is still in use for IBM mainframe computers now, but the popularity of APL has shrunk since the 1980s partially because of the difficulty to migrate it to the environment of general purpose computers .

2.4.2 Z Programming Language (ZPL)

ZPL is a parallel array programming language that was originally called Orca C. It was introduced by Washington University between 1993 and 1995 [110]. It is a data parallel language that supports most primitive operators and data types as well as complex numbers

2 Background

and parallel arrays. The main feature of this language is its flexibility in running programs on both sequential and parallel machines without the need for having any forms of explicit parallelism such as preprocessor directives [111]. A ZPL program structure is like that of Pascal programs. It is constructed of procedures that have similar forms to those in Pascal and C. A ZPL procedure can accept values or references as arguments and return a single value. The following prototype shows the general form of a ZPL procedure declaration:

```
procedure pName(argument):returnType
```

ZPL has two unique data structures, regions and directions, to declare parallel arrays and to provide indices for array references. What follows looks at these two structures and a number of array operators.

1. Regions

A region is a composed object or entity that accommodates data structures such as arrays. It also reduces the use of loops and indices, as we shall see shortly, to manipulate arrays, and most importantly helps compilers to generate code that can be executed on single-core machines or parallel machines [40]. A region is much like a conventional array but without associated data type [110]. The next statements illustrate how to use the keyword `region` to declare regions of different dimensions and then use declared regions to define region specifiers and declare arrays. The following example declares region `X` of one dimension with a set of indices $\{(-2), (-1), (0), (1), (2)\}$

```
region X=[-2..2]
```

while the next statement defines `Y` as a 2D region of size $m \times n$.

```
region Y=[1..m,1..n]
```

A region's name enclosed in square brackets (`[. .]`) is used to define a region specifier. Region specifiers are like types in Pascal, they can be used to declare arrays with the same size and rank or to augment arrays with borders. For instance, the following statement declares three single-precision floating point arrays variables `A`, `B` and `C` of type `Y`.

```
var A,B,C:[Y] float
```

2. Directions

```

direction north = [-1,0] One position "above" in relative
orientation
direction south = [ 1,0] One position "below" in relative
orientation
direction east   = [ 0,1] One position "to the right" in
relative orientation
direction west   = [0,-1] One position "to the left" in relative
orientation

```

Figure 2.3: ZPL Directions Operators

Directions can be used to declare constant vectors that refer to relative positions in a given array. This mechanism is similar to the transformation process. The constant vectors should correspond to the four cardinal directions. The general syntax for a Direction is as follows:

$$\text{direction } \textit{directionName} = [d_1, d_2, d_3, \dots, d_n];$$

where d is a constant integer called offset and n is the rank of the direction. The constant value of " d " could be positive or negative. A positive value (offset) refers to an element with higher index in that direction, and a negative value (offset) refers to elements with lower index values in that direction; however, if the d 's value is zero, it means that the entire interval for that direction will be copied into the new region or array [40]. Examples:

A direction and a base region can be combined by using an (of) operator to define (or augment) a new region adjacent to the based region. The general form is

$$[D \text{ of } R]$$

where D is a direction and R is a base region. For example, the successive effects of the following statements are as follows:

```

Line1: region X = [1..2, 1..3]
Line2: [X] A := 1.0
Line3: direction E = [0, 2]
Line4: [E of X] A := 2.0

```

Line 1 declares a 2D (2 x 3) region called X. The region specifier X then was used in Line2 to declare array A and was initialized to 1.0. After that, a direction called E is

2 Background

declared in Line3. In Line4, the *E* direction was used to augment array *A* with two columns on the right hand (east) side, and these augmented columns were initialized with 2.0.

With regard to the boundary problem, in ZPL region specifiers and the (of) operator can be used to augment arrays with borders. For example, Line4 references the 4th and 5th columns of *X*, yet array *X* as declared in Line2 did not have these columns. In this case, the ZPL compiler, according to the definition of direction*E* in Line3, extends array *X* to have another two columns on the right hand side.

3. Array Operators

- Reference operator (@)

The @ operator is used for arrays offset-referencing. It can be with an array name and a Direction to translate the indices of a given array based on the given direction vector [40]. For example, given the four directions in Figure 2.3 and a region specifier *R*, the following statement

$$[R] A := A @ north$$

increments all indices *i* in *A* by 1 where *i* represents rows indices, and it is equivalent to

$$A_{i,j} \Leftarrow A_{i+1,j}$$

While the following statement

$$A := B @ west + C @ east$$

replaces each element in *A* with the element just to its left in *B* and just to its right in *C*. It is equivalent to

$$A_{i,j} \Leftarrow B_{i,j-1} + C_{i,j+1}$$

- Reduction operator (*op* <<)

It compresses an array to form a smaller one. There are two types of reductions: Partial and Complete. A partial reduction shrinks a given array to a smaller array, and thus it requires the size of the original region (the operand) and the new region (smaller) to be specified [40]. For example, the following statement

2 Background

$$[1..m, 1] Y := + \ll [1..m, 1..n] X$$

indicates that array Y will have only one column and that the n columns of each row in X are reduced (by addition) and the scalar result is stored in the corresponding row.

A Complete reduction produces a scalar reference [40]. For example,

$$sum = + \ll X$$

results in returning the sum of all the elements in array X .

- Flood operator ($>>$)

It is similar to the concept of scalar promotion [40]. It allows you to copy elements from a low rank array to a higher dimension array. Its format is similar to the partial reduction operation. To illustrate how this operator works, assume that X and Y are two arrays with the same size and rank, the following statement

$$[R] Y := \gg [2, 1..3] X$$

then results in flooding array Y with rows that are identical to the row number 2 in array X . That is, copying 2nd row of X to each row in Y .

- wrap operator ($@^\wedge$) can be used to access to the opposite side if it attends to access outside of array boundaries.

This language is targeted at supercomputers rather than workstations. It transforms ZPL source code to ANSI C object code that can be compiled for the target machine using a C compiler. This construction gives ZPL some flexibility in linking to other languages.

2.4.3 Single Assignment C

Single Assignment C (SaC), which first appeared in 1994, is a purely functional array processing language that is designed to support numerical applications, in particular array processing [34]. Functional programming defers from imperative programming in that the computation is based on immutable objects instead of state objects, and hence purely functional languages, such as SaC, do not have side-effects in the computation process. SaC's design also focuses on supporting arrays as basic data structures such as APL and J. These two features made SaC well suited to parallel programming [101].


```

print ( reshape ( [], [1]) ) )
print ( reshape ( [3] , [3,5,8]) )
print ( reshape ( [2,3] , [6,1,7,9,5,8]) )
print ( reshape ( [2,3,1] , [6,1,7,9,5,8]) )

```

Figure 2.4: Defining Arrays in SaC

SaC is a subset of C and is also influenced by APL and SISAL. A SaC program structure is very similar to C programs which makes it easy to be adopted by programmers who are imperatively oriented. The language offers a concise way to express array operations on a high level of abstraction [34]. While most array programming languages provide built-in array operations, SaC looks at arrays as abstract data objects that have specific algebraic properties. This approach offers the flexibility to have user-defined operations that can be applied on arrays of any size and rank [112]. The following discussion give an overview of the language arrays essentials and the central WITH-loop constructor that is used in SaC to map operations on arrays.

- Array Definition

Arrays in SaC are defined using the *reshape* operator which takes two arguments: a shape vector and a data vector. The shape vector determines its rank (dimensions) and size, and the data vector supplies the initial values. Arrays, vectors and scalars are treated the same. Scalars values are considered as arrays of 0-dimension [112]. Figure 2.4 shows a simple SaC program to various arrays.

The first line defines a scalar and initials it with 1, and then outputs 1. The second line defines a vector of size three, initializes it with the given values and then outputs the result which is the vector $\begin{bmatrix} 3 & 5 & 8 \end{bmatrix}$. The third line defines 2-dimensional array of size 2×3 , initializes it with the values in associated list, and the output looks as follows: $\begin{bmatrix} 6 & 1 & 7 \\ 9 & 5 & 9 \end{bmatrix}$. The last line defines 3-dimensional array of size $2 \times 3 \times 1$ and initializes it with the same previous values, and thus the output is the same as the previous one.

- Basic Operations

- $\text{dim}(X)$ returns a scalar that represents the rank of the array X .
- $\text{shape}(X)$ returns the shape vector of the array X .
- $\text{genarray}(\text{shp}, \text{expr})$ generates an array of size shp and whose elements have the same value as expr which could be a vector.

With
Generator \implies ($\text{exp}_1 \leq \text{Identifier} < \text{exp}_2$) : value
Operation

Figure 2.5: SaC WITH loop

- *modarray(arr,expr)* defines an array of shape *shape(arr)* whose elements are set to the value *expr* which could be a vector.
- Reduction operations
 - *sum(a)* sums up all elements of the array *a*.
 - *all(a)* returns true if all elements of *a* are true otherwise returns false.
 - *any(a)* returns true if any element of *a* are true otherwise returns false.
 - *maxval(a)* returns maximum value of *a*.
 - *minval(a)* returns minimum value of *a*.
- WITH-loops

This SaC loop structure can be used to specify a given operation on a subset of elements of an array. A WITH-loop construct basically comprises two parts: a generator part and an operation part. Figure 2.5 shows the syntax of with-loop construct [34]. The first part (generator) which come immediately after the keyword WITH) defines a set of index vectors and an index variable which is supposed to present the elements of this set. The index vectors are defined by two expressions to determine the lower and upper bound of a rectangular range. Also one can initialize the selected element, in the generator part, by appending the value after a colon symbol as shown in the figure below. The second part (operation) determines the computation to be performed on each elements of the index vectors set defined in the first part.

The following example shows how a subset of array X, which is declared in the first line, is modified using the *modarray()* operation.

```

X = [9,3,5];
with
  ( [0] < K < [6] )
  modarray ( X , 8 )
```

The above with-loop example first checks the indices; K , into X , and if $K > 0$ and $K < 6$ is True, it then executes the `modarray()` function which changes or modifies the current value in the index positions; K , into 8. Therefore, the result would be [9 8 8].

The SaC compiler accepts source code as input and generates a C file which can then be compiled using one of the existing C compiler. The SaC compiler and its runtime system are capable of creating multi-threaded code that run in parallel on shared-memory architectures [34] and implicitly handle resource managements such as thread management and memory management for parallel execution.

A few years ago, Glasgow University and EPSRC had collaborated together to merge SaC and Vector Pascal technologies. The School of Computing Science at Glasgow University team worked on translating inner loops of SaC to Vector Pascal (VP) instead of C [113] to allow Vector Pascal to parallelise those inner loops, and then compile the parallelised code to libraries which can be linked back to the main SaC program. Combining the techniques of the two languages, VP and SaC, had offered some gains in performance.

Recently, an auto-parallelising compiler framework has been introduced to generate CUDA code from SaC [114]. This approach targets the data parallel loops, WITH-loops in SaC for parallel execution. It maps the WITH-loops to CUDA kernels and performs some transformations to improve performance.

2.4.4 Fortran

It was one of the first high level programming languages. Its design concentrates on general scientific applications [115]. It was the pioneer language that allowed programmers to use high-level language instead of using a particular assembly language. The first version of Fortran, which was released around 1957, included 32 statements such as DIMENSION, GOTO, DO Loops and a number of I/O statements [116]. One year later, IBM released Fortran II that included six new statements. These new statements were introduced to support pass-by references procedural programming. IBM also developed Fortran III in 1959, but it was not released due to some machine-dependent features. This problem, however, did not prevent the language from spreading and being widely used in the 1960s. The issue of portability was augmented even more as new versions of the language, which were developed by other vendors, had the same problem [117]. The importability problem, which was partially solved by Fortran IV, led to the advent of the first standard for a programming language.

In 1966, The American National Standards Institute (ANSI) approved the first Fortran standard which materialized in two versions: Basic Fortran and Fortran. The Basic Fortran

standard was based on Fortran II but did not address the portability issue [116]. The other version, which is known as Fortran 66, was based on Fortran IV [117].

The second generation standard, which is known as Fortran 77, was released in 1978. It added features such as IF, END IF, ELSE, ELSE IF, direct-access file I/O and character data type [118, 119, 116]. This standard also changed or completely removed some of the first standard's features not often utilized [116]. During the 1980s, new implemented features, such as INCLUDE, Do WHILE and END DO statements, were added to the language but were included only in Fortran 90. Recursion procedures were also supported by some Fortran 77 compilers such as DEC but was fully supported by Fortran 90 [116].

The Fortran 90 standard, which was first proved by ISO and then by an ANSI Standard, was a major revision of the preceded version. The 90 standard maintained most Fortran 77 features and included new features such as the following [116, 120, 96]:

1. Dynamic data structures using ALLOCATABLE keyword and ALLOCATE and DEALLOCATE statements.
2. Pointers that can be used to declare and reference dynamic objects
3. SELECT ..CASE for multi-path switching. It is similar to the SWITCH statement in C.
4. Grouping procedures and data together using Modules which are conceptually, similar to units in Pascal and classes in C++.
6. A free-form language in which characters do not need to be in a specific location as used in punched card programming.
7. The length of an identifier name was extended to 31 characters instead of only 6 characters as in Fortran 66 [96, 120, 116].
8. Built-in support for array manipulation.

Basic Concepts of Arrays in Fortran

- The default arrays layout in Fortran is a column-major layout.
- Array variables can be declared as primitive data types, such as integers and reals, or as a user-defined data type.
- An array lower bound is assumed to begin with 1 unless stated otherwise.
- Arrays variables can be passed as arguments to basic intrinsic procedures, such as SQRT and LEN. Passing arrays as arguments was the only feature that Fortran 77 supported in regard to arrays.

Array Processing Functionality with arrays was a significant feature in Fortran 90 which extended the basic scalar operations to operate on arrays, such as assignment operations on entire or sections of arrays [120]. It also supported dynamic arrays, pointers and overloading operators on user-defined data types, and it introduced several array intrinsic functions such as vector and matrix multiplication, reduction, reshape, inquiry functions such as SIZE, location functions such as MAXLOC. Besides, it allowed a number of scalar operations to be applied the same way on arrays with the restriction that arrays (excluding scalars) in one expression must have the same extent. That is, array operands in one expression must conform to one another in size and shape.

Essential Array Concept in Fortran 90

- An array's *rank* is the number of dimensions. For example, a variable that has a rank zero is a scalar, vectors are arrays with a rank of one (or one dimension).
- The term *bounds* refers to the upper and lower boundaries in each dimension.
- The *extent* determines the number of elements in a dimension.
- The term *shape* refers to an array that contains the number of elements in every dimension of the referenced array.

Data Parallelism in Fortran Manipulating arrays using array operations and array functions is an inherently parallel process and because Fortran 90 supports these features, it was the first Fortran standard that provided support for data parallelism [120, 96]. It was also boosted by the additional support of the OpenMP. These features had made it a base for the next Fortran standards generations such as Fortran 95 and High Performance Fortran.

Fortran 95 presented with a slight improvement over 90 such as deallocating dynamic arrays automatically. It was also augmented by other features that can be particularly used for parallelisation such as pure procedures and the FORALL statement and construct. The FORALL statement is another form of DO-loop, but it differs from a regular DO-loop statement in that its contents can be evaluated in any order [119, 19]. This approach provides some sort of implicit data parallelism. The FORALL construct, however, can be considered as multiple FORALL statements that needed to be executed in order. Functions that are called from inside a FORALL construct must be pure, i.e., no side effects.

Co-array Fortran (CAF) is an extension of Fortran 95/2003 for parallel computing [121]. The basic concept in CAF is that a CAF program is cloned a number of times or images.

Each image has its own set of data objects, and all the copies are executed asynchronously [121].

2.5 Virtual Machines

Virtual Machine (VM) technologies have been in use for decades to get around some real machine constraints. Different virtual machines models exist to solve different problems starting from splitting up a machine into separate virtual machines, optimizing programs, creating a semi-machine independent programming language or building parallel machines from heterogeneous platforms [122] .

One model is called a system virtual machine. It is used to divide hardware resources into several identical copies of the original machine. The model was proposed during the 1960s when computer machines were very costly yet large enough to be shared by multiple users. In 1972, IBM released, for its System/360-67 and System/370 machines, a time-sharing operation system called Control Program/Cambridge Monitor System (CP/CMS) [123]. The CP component was responsible for creating the virtual machine environment, while CMS was a single-user operation system that can be shared interactively. The CP/CMS OS offered a stand-alone computer system capable of running the same software that run on the original machine [123].

Another model is called a process virtual machine. This type provides an interface between a user application and an operating system or another application and allows applications to run in the same manner on any machine [123]. The most commonly used example of this model is Java Virtual Machine (JVM). JVM allows programmers to utilize VM functions rather than using machine-dependent functions. This approach offers a portable platform for Java programmers since any Java program should work on any machine on which the JVM is installed. Another example of a process virtual machine is known as Low Level Virtual Machine (LLVM). It is an intermediate compiler layer and a language-independent framework that is designed for optimising programs at compile-time, link-time or run-time. The LLVM accepts Intermediate Form (IF) code from a compiler and outputs an optimized IF code that can then be transformed and linked into machine-dependent assembler code. The first version of LLVM, which was released by the University of Illinois in 2003, was implemented in C/C++. Currently, it can accept intermediate form code that is generated by most GCC front ends to optimize C/C++, Objective-C, Fortran source code. The low level virtual machine model has become popular after many GNU front-end compilers have been updated to support this technology. Clang is one of these new compilers that was built on top of the LLVM optimiser [124]. Glasgow Haskell Compiler

(GHC) also supports LLVM and showed performance improvement when compared to an ordinary GHC compiler. Haskell developers are also working on a new compiler that can generate code for LLVM. The new compiler is called Utrecht Haskell compiler (UHC) [125], and it supports most original Haskell features. The first UHC was released in 2009 but has not yet completed.

The other common VM model is called a Unix Model. This model combines the features of the first two models; system model and process model, into one model. The Unix OS depends basically on a series of separate processes to handle user commands, and each process should have an independent set of machine resources available.

Virtual machine designs, however, have been shifted during the last two decades toward a parallel computing environment. This technology is called a Parallel Virtual Machine (PVM). It is designed for parallel networking of heterogeneous software platforms; Windows and/or Unix machines [126, 127]. A PVM program, in general, is a set of cooperating tasks that can access virtual machine resources via a set of standard routines. The first version PVM was developed in 1989 by Tennessee University, Oak Ridge National Laboratory (ORNL) and Emory University. Since then several versions have been released. The existing PVM provides a set of functions for exchanging messages, managing tasks and resources. These functions can be used for parallelising source code manually.

Virtual machines can also employ emulation techniques to incorporate a user-level instruction set using a standard instruction set. This technique allows mapping virtual resources to a real machine resources and hiding the underlying detail of the target hardware and software. This technique was adopted in this project to emulate an innovative SIMD instruction set using virtual SIMD registers and stub routines as a virtual SIMD instruction set.

2.6 Genetic Algorithms

Genetic algorithms (GAs) are based on techniques inspired by some aspects of natural science such as inheritance, reproduction and mutation, and they are used as optimization techniques for searching large solution spaces [128]. In computer science, for example, GAs could be used in data sorting and searching, circuit design and to improve the quality of designed tools such as code generation [129]. This section looks at basic genetic algorithms principles and main genetic operators. It then introduces two preceding approaches in which GAs were used for optimising generated code.

2.6.1 Genetic Algorithms Techniques

Generally, a genetic algorithm starts with a set of solutions; a native population, to breed new generations. During each successive population, a new generation is formed by selecting the recent fitter solutions as parents. The parents are then used to breed a new solution (offspring). This breeding process usually goes through mutation operations to exhibit new offspring characteristics. A new formed population (solutions) is often expected to be better than the previous population, and the process is repeated until some conditions, such as optimal solutions or number of populations, are fulfilled .

The primarily terminologies in a genetic algorithm are:

- Individual is any possible solution to the problem.
- Population consists of a fixed number of individuals genomes.
- Genomes are fixed-length of strings of genes, and they are commonly encoded as binary string as shown in the following two stripes:

Genomes 0	1	0	1	1	1	1	0	0	1	1
Genomes 1	1	1	0	0	1	0	1	0	0	1

- The fitness function of a genome is some sort of quality measure which determines how desirable it is to have that genome in the population.
- A new generation is a population that was produced by carrying out series of computations on a current population.

2.6.2 Main Genetic Operations

- Selection

In this process, individuals are chosen from an existing population (fitter solutions) to pass to a new generation. The selection operation often proceeded by the following three steps:

- Get fitness values for all solutions.
- Normalise these values by dividing each individual's fitness by the sum of all fitness values.
- Sort the current population in descending order.

Then, it comes the selection operation. There are a number of algorithms to select the fitter solutions. Some algorithms base their selection on a given arbitrary constant. In this case, only solutions that have fitness values higher than the given value are selected. Other algorithms take only a certain percentage of the best individuals. The optimiser, which will be discussed in Chapter 7, is based on the later approach; it considers $2/3$ of the best solutions in a population and retains them unchanged in the next generation.

- **Reproduction**

This step comes after the selection process. The reproduction of a new organism from a pair of solutions “parent” gives rise to offspring. This operation basically results in combinations of genes that differ from those of either parent.

- **Mutation**

This operation alters a few gene values in chromosomes or genomes from its original state [128, 129, 130]. There are several types of mutation operators such as bit string, flip bit, boundary, uniform and non-uniform. The flip bit types inverts the values in a chromosome. For example, the mutation operator for binary genes results in altering 0 to 1 and 1 to 0. However, the number of values to be changed or the mutation probability should be set low to ensure that the search will not become a typical random search [128].

3 The Cell Processor & Vector Pascal

This chapter describes the targeted processor and programming language in this research. It starts by giving background on the PowerPC architectures and the machine-dependent features such as stack frames and function calling conventions. After that, it introduces the Cell processor and its heterogeneous cores and the main components of its accelerators, and the last section in this chapter introduces the Glasgow Vector Pascal language.

3.1 Introduction to PowerPC Architectures

PowerPC stands for Performance Optimization With Enhanced RISC Performance Computing. Some of the well known RISC processors are: DEC alpha, ARC, MIPS, SPARC, and PowerPC [131, 92]. This section gives an overview of the two instruction set architectures, CISC and RISC. It then presents the main features of the PowerPC architecture and discusses the development of the PowerPC back end compiler and some machine-dependent features such as stack management and function calling conventions.

3.1.1 Instruction Set Architectures

Before the RISC architecture was introduced, many computers were based on Complex Instruction Set Computers (CISC) design. The design of CISC chips has been emphasized basically on hardware and complex instructions [131, 3]. The aim of CISC processors is to use as few lines of code as possible by augmenting more information in a single instruction which results in a complex instruction set, and hence the responsibility of handling such complexity is laid mainly on the processor hardware. This approach requires less effort to map a high-level language statement into assembly instructions. For example, the code given in Figure 3.1 shows first a Pascal statement and then the corresponding single CISC assembly instruction. The Pascal statement adds two values, B and C, and stores the result in A.

```
A := B + C ;  
ADD A , B , C;
```

Figure 3.1: Pascal Code and CISC Assembly Instruction

```
A := B + C ;  
LOAD B  
LOAD C  
ADD B,C  
STORE A,B
```

Figure 3.2: Pascal Code and RISC Assembly Instruction

Contrarily, RISC chips, like the targeted processor, design emphasizes software and reduced instructions. RISC instructions are short and simple, and therefore most of them can be executed within one clock cycle [131]. For example, Figure 3.2 shows the same Pascal statement and the corresponding RISC instructions.

Notice that the same statement was mapped into 4 RISC assembly instructions while in CISC only one instruction did the job. The above two simple examples show that RISC basically required more instructions than CISC, yet each RISC instruction does one task instead of the CISC multiple tasks.

The RISC architectures are also called Load/Store architecture since accessing memory is only allowed through the load and store instructions while other operations are carried out using registers. The advantage of register-to-register operations is to maintain reused operands in registers instead of accessing memory again and again. The other characteristics of RISC instructions are:

- Uniform instruction length
- Few machine instructions
- Most instructions are single-cycle execution.
- Suitable for optimization techniques

The disadvantage of RISC is its code size. As shown in the two previous examples, source code for RISC processors are encoded into more machine instructions than the CISC.

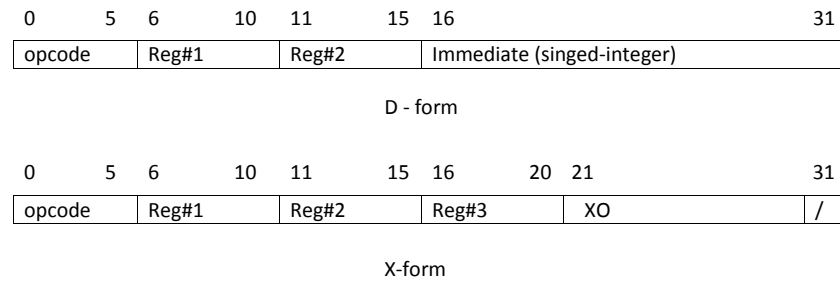


Figure 3.3: PowerPC Instruction Format

3.1.2 Main Features of PowerPC

- PowerPC instructions are fixed-length 32-bit instructions even on many 64-bit PowerPCs.
- It provides several Instruction formats, the two commonly used formats are D-form and X-form; see Figure 3.3.

An D-Form instruction can address two registers (source and destination) and contains a 16-bit immediate field. The D-form format is the only format that PowerPC architectures provide to use as an offset or an immediate value, and hence, loading a 32-bit immediate value into a register requires at least 2 instructions. For example, the common way to load an address into a register is to use first the load immediate shifted (lis) instruction which loads a 16-bit value in the destination register and then shifts it into the highest 16 bits. The second instruction is an OR Immediate (ori) instruction which bitwise ors the lowest 16 bits into that.

Figure 3.3 also shows the X-form instruction in its basic form. This form of instructions addresses two source registers and one destination register. The XO field is an extended opcode field, and the last bit can be used to alter the condition register.

- Memory is arranged as a linear array of bytes indexed from 0 to $n^2 - 1$ where n represents the execution mode of the CPU. Recent PowerPC processors provide two execution modes, 32-bit and 64-bit.
- A big-endian mode is where the most significant byte of a data item is stored first.
- Figure 3.4 shows a set of PowerPC registers. The registers are divided into three classes:
 - Volatile registers can be freely used at all times.

3 The Cell Processor & Vector Pascal

Register Type	Register No	Status	Used for
General Purpose (GPR)	GPR0	Volatile	
	GPR1	Dedicated	Stack Pointer
	GPR2	Dedicated	Table of Contents pointer
	GPR3	Volatile	1 st Argument/Return value
	GPR4-GPR10	Volatile	2 nd – 8 th Arguments
	GPR11-GPR13	Volatile	Special tasks
	GPR13-GPR31	Non-volatile	
Floating-Point (FPR)	FPR0	Volatile	
	FPR1-FPR13	Volatile	1 st -13 th Arguments
	FPR14-FPR31	Non-volatile	
Condition Register (CR)	CR0:CR1	Volatile	
	CR02:CR4	Non-volatile	
	CR5:CR7	Volatile	
Special Purpose Registers	LR	Volatile	Link Register; return address
	CTR	Volatile	Counter Register

Figure 3.4: PowerPC ABI registers conventions

- Non-volatile registers should not be altered across routines calls. However, if there is a need to use them in a called routine, then they must be saved before being used and restored prior to return.
- Dedicated registers are designated for certain tasks and must not be modified.
- Four classes of instructions
 - Fixed-point instructions that can operate on doubleword, word, halfword, and byte operands.
 - Floating-point instructions can operate on single-precision and double-precision floating-point operands.
 - Load/ Store instructions support loads and stores of fixed-point between memory and general purpose registers. They also support loads and stores of floating-point operands between memory and floating-point registers.
 - Branch instructions
- Computational instructions are not permitted to access storage. For example, to perform an operation on an operand, the contents of the operand must be first loaded into a register, updated, and then stored back in the target location.

3.1.3 Machine Code Description

A machine description process involves defining and managing a target machine's resources such as memory, registers and their types, address modes, instruction patterns and operations. Machine descriptions are used for automatic derivation of code-generator generators.

3.1.4 ILCG Notations

Glasgow VP back-end compilers use an Intermediate Language for Code Generator (ILCG) notations to map machine instructions into an assembly language. Some of the ILCG notations are:

- Memory: Accesses to memory are represented using predefined array *mem*. For example, *mem*(2000) represents the location 2000 in the memory.
- Data format: `Octect`, `halfword`, `word`, `doubleword`, `quadword`.
- Integer Data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`
- Floating-point Data types: `ieee32`, `ieee64`
- References: The keyword `ref` can be used to refer to an address of a given type. for example, `ref int32` refers to an address of a 32 bit integer number.
- Casting: Data types can be converted from one type to another using similar C syntax. For example, `(int64)ieee64` will convert the 64 bit floating-point to a 64 bit integer.
- Arithmetic operations: ILCG offers the basic arithmetic operations (+ , - , * , div , and mod). The syntax would be as follows: the operands are enclosed within brackets. and prefixed with an operation. For example, $2/(4 + 6)$ would be represented as `div(2 + (4, 6))`.
- Assignment: The assignment operator, like in Pascal, must be preceded by a colon (`:=`).

The following examples show the three basic ILCG clauses, `register`, `pattern` and `instructions`. which are used for describing machine resources. These examples are drawn from the PowerPC description:

- Register declaration

The keyword `register` is used in ILCG to define a new register and its presentation in the machine assembly.

```
register int32 R3 assembles['r3'];
register int32 R4 assembles['r4'];
...
```

Registers can also be grouped under one name as following:

```
pattern reg means [R3|R4| ... | Rn];
```

- Patterns definition

In the ILCG, the keyword `pattern` is used to define non-terminal symbols or instructions. Non-terminal symbols must be first defined before it can be used in a machine description. Non-terminal usually offer a number of possibilities or patterns that code generators can match against [42]. For example, the following code defines an non-terminal symbol called `real` which can have two possibilities; `double` or `float`.

```
pattern real means [double | float];
```

An ILCG instruction `pattern` can be used to define as an instruction that takes a number of arguments, include its semantic and the corresponding assembly code. For example, arithmetic Operations are defined as following

```
instruction pattern Op(operator op,reg r1,reg r2,reg r3, int t)
means[r1 := (t)op((t)^(r2),(t)^(r3))]
assembles [op' 'r1','r2','r3];
```

Machine descriptions should include all the instructions that are to be implemented for the target machine in a ILCG file. The ILCG file is then passed though a code-generator generator to create a Java class of the code-generator for the target machine [113]. A code-generator is used to match against patterns in the source program. If a match succeeds, the code generator will produce the assembly code associated with the matched pattern.

The ILCG notations are used to describe the sequential back-end compiler for the PowerPC and its extension. The PowerPC compiler extension includes a description of the a set of virtual registers and virtual SIMD instructions (VSIs) which are basically a set of RISC like register load, operate, store operations. The VSI set supports basic operations, such as Load, Store, Add, Sub, Sqrt ...etc, in a mapped fashion. The compiler's code generator should look at the VSIs as a set of SIMD like instruction set with a high degree of parallelism. VSIs can deploy the proper information in the right registers and invoke the proper routine. Semantics of some samples of the Virtual SIMD Instructions and how they were mapped into machine code are given in section 6.1.

3.1.5 Stack Conventions

PowerPC compilers must apply and adhere to the following stack conventions:

- The stack must grow from higher addresses to lower addresses.
- The stack pointer (SP) must be kept in the general purpose register number 1 (*GPR1*).
- The SP must point, at any given time, to the lowest address of the stack.
- Create a stack frame for every function or procedure if needed.
- A stack frame must be alive from the point the procedure is called or executed until the procedure returns to the caller or reaches its end.

3.1.6 Stack Frames Management

Exchanging information between different program routines usually requires space on the stack to keep the information required for this inter-routines communications, and therefore, for every active routine at any given time, there is often associated a stack frame. Stack frames are created during the beginning (prolog) of a procedure call to set up the new execution environment and released at the end (epilog) of a called procedure. The information that should be kept in a stack frame is as follows:

- An area to keep hold of passed parameters.
- A place to maintain a stack pointer.
- A location to save the return address.
- Area to store special purpose registers that could be altered.

- A room for local objects.
- A space that is enough to store non-volatile registers.

However, several architectures, such as Intel, offer instructions that can automatically perform these tasks, but on architectures, such as PowerPC, these two tasks or sessions must be hand coded. The following steps show what needs to be done in each session:

- Prolog (ENTRY) Session

This session establishes a new execution environment for the called routine. This establishing process involves four basic steps:

- Save the caller's frame pointer and the address of the return point.
- Compute the new stack frame size and align it.
- Allocate a new stack frame and update the SP and the FP.
- Maintain a link to nested routines. More explanation shall be given in the implementation of the PowerPC back-end compiler.

- Epilog (LEAVING) Session

The Epilog session is supposed to release a called procedure's frame once it is executed and to re-establish the old environment before returning back to the caller. This process also involves four steps:

- Restore any saved registers.
- Release the current frame by updating the SP.
- Restore the return address.
- Return to the caller.

3.1.7 Functions Calling Conventions

Functions calling conventions are part of what is called an Application Binary Interface (ABI) which defines the rules that permit separately compiled routines in the same or different languages to interact with each other. In other words, a compiler of one language must adhere to the calling conventions of other languages that the compiler expects to interact with in order to set up the right environment and to ensure safe and efficient

interaction between modules from different languages. The subsequent discussion introduces the main Pascal and C calling conventions on the PowerPC and other parameters associated with function calling mechanisms:

- Pascal Calling Conventions

- Parameters are pushed on the stack from left to right.
- Function results are returned on the stack.
- The caller locates a space on the stack for a function result before it pushes the parameter on the stack.
- The caller removes the result from the stack.

- C Calling Conventions

C programming language was chosen here because it is widely used and many other languages apply the same C rules. The C calling conventions are:

- Parameters are passed from left to right
- Arguments are passed via registers. If there is not enough registers, then the remaining arguments are stored on the stack frame.
- Scalar function results are returned in registers.
- Structured data types are returned by their pointers.

- Passing Arguments Conventions

A PowerPC compiler must conform to the following conventions when passing arguments from one routine to another:

- Parameters that are passed via registers must comply with the specified utilization of the individual registers as shown Figure 3.4.
- If the parameters to be passed need more space than the available registers can handle, the stack frame then should be used to hold the additional parameters.
- When a procedure sets up its stack frame, it has to reserve space for the largest number of parameters that a procedure call requires.

- Parameters are pushed on the stack from right to left, and hence the first parameter (from left) will be located on the top of the stack. This approach guarantees that any function can determine the exact offset of each parameter no matter how big the parameter list area is.
- Nested Procedures Conventions

Vector Pascal, unlike C, allows nested procedures in which a procedure is encapsulated within another procedure. In nested procedures the crucial point is that compilers must be able to access data in higher layers of nesting. Display, Lambda lifting, and Static Linking are some of the techniques that can be employed to handle nested procedures. Only the first technique was considered here since the PowerPC back-end compiler employed it to keep track of frame pointers of nested procedures.

The compiler uses this technique to store frame pointers of previous nested procedures which can be used with offsets to access any value in the nested procedures. This linking process is handled during the prolog session. To illustrate that, let us assume the following:

- The general purpose register ($r1$) holds the stack pointer (SP) and ($r31$) holds the running procedure's frame pointer (FP).
- The FS variable specifies the frame size for a procedure.
- The NL variable determines the lexical level of a procedure.
- The Display area is the space at the start of the current frame.

Now, the following steps explain how the Display technique works:

- If $NL = 0$ then
 - * Store the current (caller's) FP on the stack. The new frame pointer (FP) should be the same as (SP).
 - * Locate a space for the new frame by incrementing SP. The SP should be incremented by a calculated frame size.
- If $NL > 0$ then (Nested procedures)
 - * Store the current FP on the stack (display area). For the called procedure points view, this represents the proceeding nest procedure's frame pointer.
 - * Maintain a copy of the current SP which later on should be the new frame pointer.

- * Now iterate through the display area of the current frame to copy all nested procedures' frame pointers, which are already cited in the current frame, into the display area of the new frame. The loop structure should iterate NL times and in each iteration it does the following:
 1. Get the NLth nested frame pointer.
 2. Store the pointer into the display area of the new stack frame.
 3. Decrement NL
 4. IF N > 0 Go To Step 1
- * The new frame pointer FP should take the SP value to point to the start of the new frame.
- * Now, locate a space for the new frame by incrementing the stack pointer. The SP should be incremented by a calculated frame size.

- Return values conventions

A procedure result or function's return value can be considered as an output parameter. According to the PowerPC's ABI, fixed-point and floating-point return values must be returned in the general purpose register No. 3 (*GPR#3*) and in the floating point register No. 3 (*FPRG#3*) respectively.

PowerPC architectures also don't support stack operations, such as Pop and Push. Besides, they don't have hardware support of some functions such as trigonometric functions. During the development of the back-end compiler of the PowerPC, I had to code manually all these machine-dependent issues starting from coding machine description and stack operations up to handling Pascal and C calling conventions. The hand-coded implementation of these issues will be discussed in depth in the following chapters.

3.2 The Cell Broadband Engine Processor

The Cell architecture was introduced in 2006 by Sony, Toshiba, and IBM, but its first mass-market was with PlayStation 3 consoles which were released in 2007. The Cell is a heterogeneous multi-core processor that consists of a general Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs) [54, 55, 56]. The Cell processor potentially offers high levels of parallelism, but it is not easy to program due to its heterogeneity of CPUs, memory structures and instruction sets. The PPE processor is a PowerPC

architecture that is responsible for the overall control of the processors while the SPEs are simple RISC architectures that are mainly designed for computation. The first generation's SPEs performance on double-precision, however, was poor; it was around 10 times slower than its performance on single precision. For this reason, the manufacturers pushed the introduction of the second generation of the Cell processor in 2008. The second Cell generation, known as PowerXCell, was very similar to the Cell BE except that its SPEs were re-engineered to improve double-precision performance [56]. The PowerXCell's SPEs was five times faster than the first generation in handling double-precision operations. The Cell BE processors were the main architecture in the fastest computer in the world in 2008, and is still in the Top 10 positions of the 37th TOP500 List of 2011 [132].

Cell is a distributed-shared memory architecture which has main memory space on the PPE and private Local Storage (LS) on each SPE. The main memory represents the entire effective-address space that is available for all Cell's elements. The SPEs LS's can be accessed directly by the SPEs or by the PPE through DMA controllers in a non-coherent mode [56, 57, 54]. The other heterogeneity of the Cell appears on the instruction level as the PPE and SPE's have different instruction sets. The machine's heterogeneity presents a major challenge for developing Cell applications because it requires writing source code for each core type and requires two compilers.

A number of parallel programming models have been released to parallelise applications on the Cell processor. Some of these models are applying task or thread parallelisation schemes such as CellVM and Hera-JVM [48, 133]. Others focus on data parallelisation schemes such as OpenMP, SieveC++ and OffloadC++ [79, 38, 29]. Also, the recent GNU tool chain and IBM XL offer compilers for C/C++ and FORTRAN on both architectures, the PPE and the SPEs [134, 135]. The Cell architecture showed performance potential specially when using good heterogenically adapted code [56], and this is what motivated us to select this architecture.

3.2.1 The Power Processor Element (PPE)

The PPE processor is a general purpose 64-bit PowerPC architecture. It is designed to handle overall control of the system such as running OSs and coordinating the SPEs. The PPE processor consists of:

- 3.2 GHz PowerPC processor.
- 512MB Main memory
- 32 KB L1 data cache and 32 KB L1 instruction cache

3 The Cell Processor & Vector Pascal

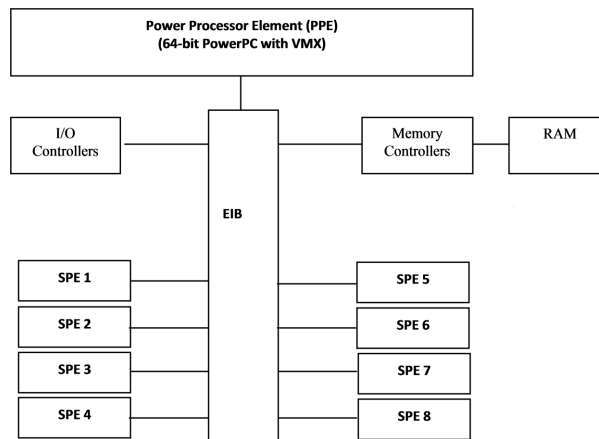


Figure 3.5: The Cell BE Schematic Diagram

- 512 KB L2 caches
- 32 x 128-bit vector registers
- One fixed-point integer unit and one floating-point unit
- Load/Store unit.
- An instruction control unit and branch unit
- Multimedia Extensions Unit (VMX)

The PPE instruction set is an extended version of the PowerPC architecture instruction set with slight changes. The PPE instruction set includes SIMD extensions and C/C++ intrinsic for SIMD extensions [56].

3.2.2 The Synergistic Processor Elements (SPEs)

The SPEs are mainly designed for manipulating data. They are independent simple SIMD RISC architectures that are very similar in function to vector processors in which a single instruction operates on multiple data elements [56]. An SPE, as shown in Figure 3.6, consists of two independent main units: a synergistic processing unit and a memory flow controller.

1. A Synergistic Processing Unit (SPU)

- a) 128 x 128-bit registers.

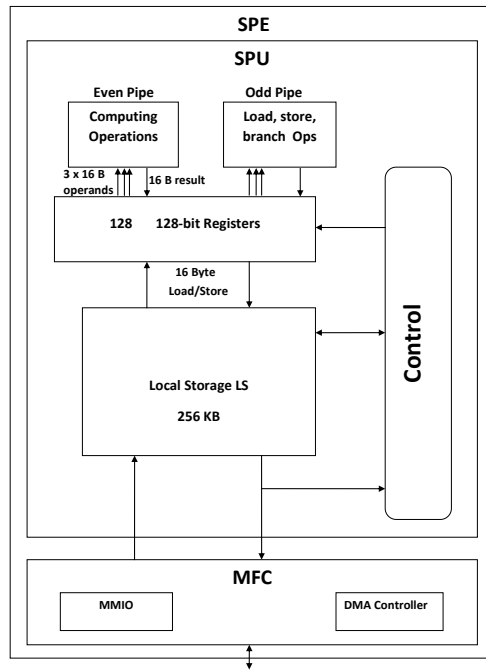


Figure 3.6: SPE Hardware Diagram

An SPU operates only on 128-bit vector data types and supports only 16-byte load and store operations. Using literal scalars will not be an efficient approach as the SPUs have to perform a 16-byte read and data shuffling to align scalars.

b) Local Storage (LS)

Each SPU has a 256KB local memory that is directly addressable by the SPU. Due to the small size of the LSs, the SPUs will be very efficient for applications in which most of the work is done on blocks of data such as in numerical processing, image processing, or video streaming.

c) Two Execution Units

Every SPU contains two parallel pipelines unit, Even and Odd, for executing instructions. The Even unit handles fixed-point, floating-point, logical and byte operations, while the Odd unit handles load, store and branch operations. The two pipeline units allow users to issue two instructions per cycle; for example, one memory instruction and one compute instruction.

2. A Memory Flow Controller (MFC)

MFC is an interface that controls most of the inter-processor communication between an SPU and the other processor's components. Because every SPE has its

own MFC, this permits the SPU to work on different system memory spaces concurrently. The interdependency between an SPU and its MFC also allows data to be transferred in parallel while an SPU is executing other instructions [91]. An MFC is built of:

a) Memory Access Controller

These controllers are responsible for data movement within the Cell processor [56, 54]. A memory controller includes a Memory Management Unit (MMU) that handles the mapping of system addresses and two queues that accommodate DMA transfers issued by the PPE or the SPEs. One queue is for PPE-initiated DMAs and can hold up to 8 DMA requested. The other queue is for SPE-initiated DMAs and can hold up to 16 DMA requests.

b) Memory Mapped Input Output (MMIO) Registers

The MMIO control registers, which are primarily used to map memory addresses, can be used by the PPE to load an SPE program and data. They can be also used by one SPE to access the local storage of other SPEs [56].

c) Atomic Unit (ATO)

The ATO unit handles atomic DMA commands (functions) and atomic operations [56]. Atomic DMA functions are more reliable in handling shared data than regular DMA functions which can be easily interrupted [91]. With atomic functions such as `getllar()` and `putllac()`, a user can set a lock on an aligned unit of storage. On the Cell processor, the unit of storage is 128 bytes that is aligned to a 128-byte boundary. The VM model depends on these two atomic commands to synchronize operations such as store and reduction. I will elaborate on this point as we discuss the synchronization techniques in the coming chapters. Atomic operations are relatively quick compared to locks, and do not suffer from deadlock and convoying. The disadvantage of the Cell atomic operations is that they only do a limited set of operations on a limited space of bytes.

3.2.3 Cell Memory Space

The Cell processor is a distributed-shared memory architecture that has eight private local spaces and the main memory space. An SPE local storage is a memory space that can be accessed only by its SPU [136, 54, 91, 56]. The PPU main memory represents the entire

effective-address space that is available for all Cell processor's elements. The PPU can access an SPU's LS through DMA controllers in a non-coherent mode. The main memory space can be also configured by the PPU to be shared by the SPUs through the MMIO registers of the MFCs [56].

3.2.4 MFC Interfaces

An MFC communicates with its SPU through two interfaces:

1. SPE Channel Interface

Each MFC has 32 channels to communicate with its SPU. The channels offer a local communication within an SPE and have low latency specially for non blocking commands [56]. The channels can be configured as a one-way communication; read-only or write-only or blocking or non-blocking channel. Each channel has an associated channel count that indicates the outstanding operations. The three main SPE channel instructions are:

- Read Channel (*rdch*)

The *rdch* instruction can be used only on channels configured as read channels. For example, the following statement

rdch r5, chNo

reads data from channel “*chNo*” into the general purpose SPE register “*r5*”.

- Write Channel (*wrch*)

The *wrch* instruction is similar to Read instruction. It can be used only on channels configured as write channels.

- Read Channel Count (*rchcnt*)

The *rchcnt* instruction reads the channel account (zero or one). For example, if a given SPE channel capacity is zero, then the SPE must be stalled until the channel count changes to >0. The stall overhead is high, and therefore one should try to avoid such stalls.

2. MMIO Interface

The MMIO interface can be used by both the PPU and SPUs to communicate with any MFC.

3.2.5 Cell Communication Mechanisms

- Direct Memory Access (DMA) Transfers

- DMA Commands (functions)

A Cell's transfer command basically is a function with several parameters. The most important four parameters are: local store address, main memory address, size of data to be transferred in bytes, and a tag. Both local store and main memory addresses must be aligned on 16-byte boundaries. For a better performance, addresses should be aligned on 128-byte boundaries[91]. These alignment constraints are very crucial when accessing main memory because they most likely will introduce problems associated with data sharing. A single DMA command can be a single data transfer that ranges from one byte up to a stream of 16-KByte [91]. One can also issue a list of DMAs in a single command. The list can hold up to 2048 single data transfers, and hence it can transfer streams of up to $2048 \times 16K \approx 32MB$. The VM model uses a single data transfer aligned on 128-bytes boundaries to load/store data from/into the main memory. The tag is used to identify a DMA using values ranging from 0 to 31 and can be used to group an individual DMA by giving them the same tag value.

The Cell processor provides two functions (commands) for transferring data. One function is called *GET*. It transfers data from the main memory to an LS. The other is called *PUT*, and it transfers data from a LS to the main memory. Affixes, such as “*f*”, “*b*” and “*l*”, can be added to these two functions to oblige some conditions on the transfer process [91, 56, 54]. The affix “*f*” means Fence synchronization, “*b*” means Barrier synchronization and “*l*” implies that a function operates on a List of DMAs. The Cell processor also provides other functions that can be used to check a DMA completion status, synchronization purposes,...etc.

- DMA Transfer Process

A transfer request can be initiated by either the PPU or the SPUs. Transfers issued by the later, called SPU-initiated DMAs, are faster than DMAs that are issued from the PPU due to the following:

1. The PPE does not have an MFC, and so it depends on the target SPE's MFC to act on its behalf to transfer data [54, 91].

2. The PPE command queue can house only 8 requests while the SPE queue can hold up to 16 requests.
 3. The PPE can issue only one DMA command at a time, but the eight SPEs can dispatch eight DMA commands simultaneously.
- Mailboxes

Mailboxes are FIFO queues that can be used to exchange 32-bit messages between the processor units (PPU and SPUs) [56]. This mechanism is useful to exchange short messages or control data. Each SPE has two types of mailboxes: Inbound and Outbound mailboxes:

- Inbound Mailbox

An Inbound mailbox is a 4-entry FIFO queue. Inbound mailboxes are used to send messages from the PPE to the SPEs and between the SPEs.

- Outbound Mailbox

Each SPE has two Outbound mailboxes: Outbound mailbox and Outbound Interrupt mailbox. The outbound mailboxes are 1-entry queue. They are used to send messages from one SPE to the PPE or to other SPEs.

The SDK library provides a number of functions to access mailboxes easily [135]. The SDK provides two classes of functions: *spu_*_mbox* functions and *spe_*_mbox* functions. The first class can be used by the SPEs to read, write or check the status of a mailbox, while the second class can be used by the PPE to read, write or check the status of a mailbox. The PPE SDK functions for accessing mailboxes are non blocking functions by default, and the SPE SDK functions are blocking functions by default [56]. Thus writing from the PPE side to a full SPE inbound mailbox may result in overridden previous messages, and on the other hand, reading from an empty mailbox or writing to a full mailbox by the SPUs result in stalling the SPU. To avoid overridden previous messages or stalling the SPU, a user must explicitly check the status of the target mailbox before attempting to use it.

- Signals

Signaling mechanism can be also used to signal messages between the PPE and SPEs. Each SPE has two 32-bit signal registers: Signal Notification 1 and Signal Notification 2.

The main difference between DMA mechanism and the other two mechanisms is that with DMAs, an SPU hands its requests to its MFC to transfer data between memory locations, but with mailboxes and signals, an SPE can only write messages to a mailbox hoping that the intended recipient will recognize it and read it.

The VSM implementation depends on DMAs to move data between the main memory and LSs and depends mostly on mailboxes to exchange short messages such as passing the starting addresses of data blocks to the SPEs or sending acknowledgments between the PPE and the SPEs.

3.2.6 Exchanging Messages Using SDK functions

The PPE can send messages to an SPE inbound mailbox using the *spe_in_mbox_write* function, and the SPEs can read messages dispatched in their inbound mailboxes by using *spu_read_in_box* function. The *spe_in_mbox_write* function, however, is slower than using the MMIO registers, which shall be discussed shortly, because it involves a system call [74, 56].

An SPE can also send a message to the PPE or other SPEs by dropping a message in one of its outbound mailbox using *spu_write_out_mbox* or *spu_write_our_intr_box* functions. Once the message is dropped, the receiver can then read the sender outbound mailbox.

3.2.7 Exchanging Messages Using MMIO Interface

- PPE to the SPEs

The MMIO interface can be also used to exchange messages between the PPE and the SPEs or between the SPEs by writing directly to corresponding SPE's MMIO registers. In order for the PPE to communicate with an SPE, it must first map the corresponding problem state area of the SPE to the PPE address space by using the *spe_ps_area_get* function. The function is part of the SPE libraries "libspe2.h" offered by the manufacturers [56], and its prototype is as follows:

```
void * spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)
```

The first argument (*spe*) is an identifier of an SPE thread, and the second argument is a pointer of type problem state area (*ps_area*) which needs to be mapped on the PPE address space. The function returns a pointer to the requested problem state area. There are a number of problem state values for the second argument

```

typedef struct spe_spu_control_area {
unsigned char reserved_0_3[4];
unsigned int SPU_Out_Mbox;
unsigned char reserved_8_B[4];
unsigned int SPU_In_Mbox;
unsigned char reserved_10_13[4];
unsigned int SPU_Mbox_Stat;
unsigned char reserved_18_1B[4];
unsigned int SPU_RunCntl;
unsigned char reserved_20_23[4];
unsigned int SPU_Status;
unsigned char reserved_28_33[12];
unsigned int SPU_NPC;
} spe_spu_control_area_t;

```

Figure 3.7: SPE_CONTROL_AREA Structure

(ps_area), but the most important one which will be used by the VSM's messaging protocol is called `SPE_CONTROL_AREA`. By passing `SPE_CONTROL_AREA` as the argument (ps_area) to the function `spe_ps_area_get(...)`, the function returns a pointer to a control area structure of the specified SPE (spe). The contents of the control area structure are shown in Figure 3.7.

Once the problem state area is mapped, the PPE can then directly access that SPE control area by using the pointer of the `SPE_CONTROL_AREA` structure as a base address to poll or update the parameters that are defined as members of the structure. See Figure 3.7.

Using the MMIO registers does not involve the kernel (a system call) and therefore it is more faster than using SDK built-in functions. For this reason, the VSM model depends mainly on MMIO registers to forward messages from the PPE to the SPEs.

- SPE to SPE

The first step that is required for the SPEs to communicate with each other is similar to above. The PPE code must first map the controls area of the SPEs to the PPE address space. The second step is different because the PPE must notify each SPE with the pointers of the control area of the other SPEs, and once an SPE knows the pointer of the control area of the receiver, the SPE can then use a regular DMA transfer to access the receiver's mailbox.

3.2.8 Managing Threads on the Cell

- Pthreads

The main function or procedure that runs on the PPE is the main thread of type pthread. The main function's return implicitly triggers a call to `exit()` using the return value from the main functions as the exit status. pthread can also be used to create threads that run on an SPE. The following function synopsis shows the elements needed to create a new thread. The first argument in the function holds the ID of the created thread. The second argument is often set to NULL. The third arguments determine the entry location of the function to be executed and the last argument represents the `start_function`. When the executed `start_function` returns, the effect would be as if the `pthread_exit()` function is called using the same value that the executed function returned.

```
INT PTHREAD_CREATE(PTHREAD_T *ID, PTHREAD_ATTR_T *ATTR,
VOID*(*START_FUN)(VOID*), VOID *ARG);
```

- SPE Context (thread)

Cell provides several functions to manage SPE threads and load code on the SPE's. The list here shall focus only on the main functions and parameters which will be uses in the VSM implementations.

```
- SPE_CPU_INFO_GET(UINT FLGS, 0)
```

This function can be used to get information on several issues depending on the `flgs`'s value. For example, with the `SPE_COUNT_PHYSICAL_SPES` flag, the function returns the number of SPEs available on the whole system. With the `SPE_COUNT_USABLE_SPES` flag, which the current implementation of the VSM uses, the function returns the number of SPEs that are available at this point in time.

```
- SPE_CONTEXT_CREATE(UINT FLAGS, NULL);
```

This function creates a new SPE context. If the function succeeds it returns a pointer to the created context. The VSM uses the `SPE_MAP_PS` flag in order to get permission for memory-mapped access to the problem state areas of the SPEs. This feature is mainly used to get access to MMIO registers to exchange messages between the PPE and the SPEs.

```
- SPE_PROGRAM_LOAD(SPE_CONTEXT_PTR_T , SPE_PROGRAM_HANDLE_T *)
```

This function loads an SPE main program into its local memory. The first argument must be a valid pointer to an SPE context for which the program is loaded. The second argument determines a valid address of the SPE program to be loaded. This step must come before launching an SPE context with `spe_context_run`.

– `SPE_CONTEXT_RUN(SPE_CONTEXT_PTR_T, UINT*, NULL, VOID*, VOID*, NULL)`

The first argument must be a valid pointer to an SPE context that needed to be run. The second argument determines the entry point from which the SPE program starts executing. The `SPE_DEFAULT_ENTRY` flag can be used to start the execution from the first statement in the SPE main program. It is important to notice that the `spe_context_run` function is a thread blocking call which means any thread whether a PPE or SPE thread that calls `spe_context_run` to launch another an SPE thread will be blocked until the called SPE thread terminates and returns. This is a very critical issue when it comes to the PPE because if the main PPE thread launches an SPE thread via `spe_context_run` the main thread then has to wait for the new launched SPE thread to finish. I solved this problem by using separate pthread threads to call `spe_context_run` to unblock the main PPE thread from communicating and exchanging messages with the SPEs.

3.2.9 Programming Cell Applications

Programming the Cell multicore processor presents a new challenge due to its instruction set heterogeneity which requires dual source code: a master program and a child program. The master program runs on the PPE, and its overall role is instantiating a main thread, spawning SPEs threads, and handling I/O requests. The generated code corresponding to the master code should be very similar to other PowerPC architectures [56]. On the contrary, the child program runs on an SPE, and its overall role is to handle data transfers between the main memory and LSs and most importantly to perform computation.

Currently, the two programming languages that are fully supported by the Cell processor's manufacturers are C/C++ and FORTRAN. The manufacturers provide also a Software Development Kit (SDK) that includes tools for writing Cell applications, debugging tools, optimized library routines and Eclipse IDE [56]. Some of the tools that are available now are:

- The GNU toolchain for the Cell processor provides C/C++ and FORTRAN compilers, assembler and linker for both PPE and SPEs. These two languages support

OpenMP. The tool chain also includes an Ada language compiler but for the PPE only.

- IBM XL C/C++ and FORTRAN compilers for Linux. The XL compilers generate code for the two architectures, but they require the GCC tools for assembling and linking programs for the PPE and the SPEs. Both compilers support OpenMP.
- A SPE runtime management library (*libspe*), SIMD math library and other mathematical libraries.
- A Data communication and synchronization (DaCS) library to handle communications between Cell components.

The following steps show how to build a Cell application written in C++ using the GNU toolchain:

1. Compile the child program “*speProg.cpp*”, *speExec* is the SPE binary

```
spu-g++ speProg.cpp -o speExec
```

2. Embed the binary file “*speExec*” into a PPE compatible object format. The first name (*speProgram*) is the name that the PPE is expected to use to call the SPE binary.

```
ppu_embedspu speProgram speExec spe2ppeFormat.o
```

3. Compile the master program “*ppeProg.cpp*” and link it with the object file.

```
ppu_g++ ppeProg.cpp spe2ppeFormat.o -lspe -o cellExec
```

3.2.10 Executing Cell Applications

The execution of a C/C++ application on the Cell processor is mastered by the PPE. Figure ?? shows the main execution steps involved in launching an SPE program. Once a Cell program is executed, the compiler automatically creates a main thread on the master processor, and then goes through the following steps:

1. loads the main thread with the PPE program and then starts the execution of the application from the `main()` function.
2. The PPE program should then create SPEs threads and issue a command to the MFCs of the SPEs to load the SPE program from the system memory into the SPEs’ LSs.

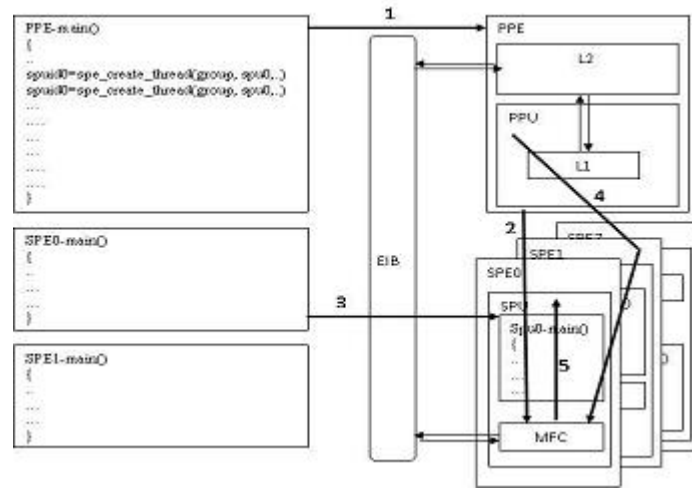


Figure 3.8: SPE Hardware Diagram

3. Each MFC then starts loading the code from the main memory into its SPE LS.
4. The PPE also requests from the SPEs' MFCs to begin executing the loaded thread.
5. The requested MFCs in turn orders their SPUs to start the execution of the threads.

Though programming the Cell is not an easy task due to its heterogeneity of memory structures and instruction sets, it showed performance potential specially when using good heterogenic-code. This was, in fact, one of the motivations which led to investigate the possibility of developing a programming tool that can help Cell compiler developers and simplify the development of Cell parallel applications.

3.3 Glasgow Vector Pascal

Vector Pascal (VP) is an extension to the standard Pascal programming language. There are three different implementations of Vector Pascal. The first implementation was introduced by Turner [137] in 1987 at Iowa State University, US. In 1992, Formella [138] developed another version at Saarlandes University, Germany, and in the early 2000's, Dr. Cockshott and his supervised students developed an independent version of Vector Pascal at Glasgow University, UK. The attempt here is to port last version to the Cell processor, and thus the term Vector Pascal shall, henceforth, refer to Glasgow's implementation [139].

Glasgow Vector Pascal (VP) extended standard Pascal by supporting SIMD instruction set extensions and data parallel operations [58, 42, 7]. Glasgow VP extension is based on Kenneth Iverson's notations to provide an concise notation to express data parallelism.

Iverson's approach, which was introduced in 1962 to address data parallelism, is was a machine independent approach that can be safely used to express scalar instructions or array instructions [35]. The approach presented also answers to solve array issues such as how to deal with operations on arrays of different ranks and sizes.

Glasgow's extension also introduced new data types such as pixels and new operators such as Input and Output of arrays, Min and MAX to find the smallest or the largest elements in an array. In VP, operations can be applied to arrays in the same way that they could have been applied to scalars leaving the compiler to decide whether the operation is an array or a scalar operation based on the declaration of the operation's operands. These features make it a suitable language for the expression of data parallel. The language has also introduced new notations and overloaded some standard Pascal operators in order to exploit data parallelism, and it has recently been extended by the introduction of the PURE keyword for parallelisation purposes [140].

A number of back-end Vector Pascal compilers for common architectures had already been developed at Glasgow University [7, 140]. The code generator for each machine is a Java class that is automatically generated from a formal machine description written in Intermediate Language for Code Generators(ILCG). An automatic code-generator generator translates the machine specification, written in ILCG, into an optimizing code generator written in Java.

This language was targeted because first the aim is to used an array programming language that supports features which implicitly express data parallelism such as arrays operations. Secondly, array names in VP can be safely used in arithmetic operations to operate on data. Thirdly, VP was developed at Glasgow and more support is available such as the source code, and the main developer. Yet, the most important feature is that the VP front-end compiler is already designed to support SIMD technology in flexible degree of parallelism mode, and hence its code generators can be easily switched to produce look-alike SIMD code that operate on large registers. Moreover, the manufacturer provides full C/C++ and FORTRAN support for programming the Cell processor, and there are another two attempts, CellVM and Hera-JVM, to support Java on the Cell processor.

The following Pascal code segment illustrates how operations can be applied to arrays in the same way that could have been applied to scalars:

In Figure 3.9, the first line declares *v1*, *v2* and *v3* as arrays of size 1024 of type integer. The third line is an additional operation that involves all the elements of *v1* and *v2*. This statement results in adding each element in vector *v1* to the corresponding element of vector *v2* and storing the sum in the corresponding element in vector *v3*. Line 3 in Figure 3.9 shows a simple example of an array expression that a compiler can safely chop it into block and then process in parallel.

```

Line1: var v1,v2 , v3: array[0..1023] of integer;
Line2: begin
Line3:   v1 := v2 + v3 ;
Line4: end;

```

Figure 3.9: Pascal Code Segment

3.3.1 Vector Pascal Features

VP currently supports a number of operations such as reduction, conditional update operations, array reorganization, permutation, unary operators, and dimension types ... etc. Some of these new features in brief are:

- Pixel Data Type

This new data type is commonly used in image processing. Pixels are implemented in Vector Pascal as 8 bit signed integers that are dealt with as binary fractions.

- The Reduction Operation

The reduction operation takes arrays of rank m and returns an array of rank $m - 1$. The interpretation of reduction for commutative operators is simple. Consider:

```
s := \ + v;
```

where s is a scalar and v is a vector. The left-hand side returns the sum over the vector and the result is assigned to x .

This operator can also be used to calculate the dot product of two vectors. For example,

```
x := \ + (v1 * v2)
```

However, the dot product of two vectors can be written in standard Pascal as the following:

```

for i := 0 to n do
    s := s + v1[i] * v2[i] ;

```

where n is the smallest rank among the arrays v_1 and v_2 .

- Assignment Operators

Vector Pascal extended the Pascal array assignment operator to handle operations on different rank expressions. For example, the following Vector Pascal code segment,

```
arr : array[0..4, 0..4] of integer ;
vec : array[0..4] of integer ;
vec := 5 ;
arr := vec * 2 ;
vec := \ + arr ;
```

results in assigning 5 to every element in vector “vec”, and then assigning 10 to all elements of array “arr”. The reduction “\ +” operator in the last statement results in summing all the elements of each row in array “arr” and then assigning the result to the corresponding element in the vector “vec”. To write the same code segment in Pascal requires more than 12 lines of code [35]. VP also allows mixed rank expressions by depending on the VP compiler to automatically generate a loop that can span the ranks of the distension and the source operands, and then the evaluation is carried out based on the number of dimensions of the array on the left-hand side.

- Slice Operation

It is a useful operation for many applications that need to manipulate specific sections of arrays. VP has overloaded array abstraction to define sub-ranges of arrays. A sub-range of an array can be defined by using the array name pursued with a range expression. For example, the following statement

```
v1[2..4] := v2[1..3] * 2 ;
```

results in doubling the 2nd – 4th elements of vector v_2 and assigning the values to the third, fourth and fifth elements of v_1 .

- Dyadic Operations

Arithmetic and logical operations such as -, *, /, div, mod, <, >, <=, >=, =, <, >, and, or, shr, shl, min, max, are overloaded by VP to operate on arrays. VP also

supports dot product vectors and matrix multiplication by using the dot “.” operator as illustrated in the following code segment:

```
a1, a2 : array[0..4, 0..4] of integer ;  
v1, v2 : array[0..4] of integer ;  
v1 := a1 . v2 ;  
a2 = a1 . a1 ;
```

results in transforming vector $v2$ by the matrix a and performing matrix multiplication .

- Loop Unrolling

Arrays are suitable data structures that can benefit from vector processing because one can perform an operation on a group of elements or sub-range of an array at a time instead of on individual array elements. The array slicing technique helps to implicitly define vector operations by unrolling array operation. For example, let $v1, v2$, and $v3$ be vectors each containing 12 real elements, then the following statement

```
for j := 1 to 12  
  v3[j] := v1[j] + v2[j];
```

adds one element from $v1$ to an element in $v2$ and assigns the result to the corresponding element in $v3$. This loop which adds the vectors element by element could be, however, unroll using sub-ranges based on the data type. The above loop structure could be unrolled as follows:

```
for j := 1 to 12 step 4  
  v1[j..j+3] := v2[j..j+3] + v3[j..j+3];
```

to suit, for example, an SIMD instructions set that operates on 128-bit registers given that the real elements each require 32-bit.

```

pure function foo(int: i):integer;
begin
  foo:=i*i;
end

```

Figure 3.10: A Simple VP Pure function

3.3.2 VP Compilers

A VP front-end compiler, which is a machine-independent compiler, had already been developed at the University of Glasgow [113, 141, 142]. Also a number of back-end compilers had already been developed at the University of Glasgow for common architectures such as Intel's Pentium series, AMD-Opteron and the Sony emotion engine used in the PS/2, but not for the PowerPC architectures [113, 141]. Early versions of the compiler supported only single core machines using the MMX and 3DNow parallel instruction sets. VP also supports [140] the new Advanced Vector Extensions (AVX) instruction format that supported by processors such as Intel Sandy Bridge and AMD Bulldozer [86, 143]. Subsequently parallelism has been extended to allow array expressions to be evaluated across multiple cores. Recently, the language has been extended to allow the use of PURE functions for parallelization purposes [140]. Pure functions can be defined by adding the keyword PURE before the function name. A VP pure function must be side effect free to potentially permit mapping it over array arguments to be performed in parallel. Figure 3.10 shows a simple VP pure function.

A VP back-end compiler implementation basically includes the following:

- Machine description

A machine description defines available registers, type of registers, instruction patterns and basic operations. The instruction patterns determine the mapping of the machine semantics into assembly instructions.

- Machine-Dependent Routines

Different architectures have different features that are often handled on individual bases to ensure the reliability between different environments. For example, function calling mechanisms on PowerPC architectures must be handled explicitly by compilers.

3.3.2.1 Building a VP Compiler

- Translate the machine description into *.ilcg format using m4 macro processor.
- Generate a java file from the targetMachine.ilcg file using ILCG. The ILCG language parses *.ilcg file and generates a Java method for each defined operation in the parsed file.
- Compile the generated Java code along with the manually written routines to create the corresponding classes.
- Combine all the classes include the front end classes in Java archive file format.

3.3.2.2 Compilation Process

The compilation process is divided into three main steps:

- Compile Source Code
 - Pares a Vector Pascal source file using a hand written recursive descent parser to generate an internal data structure; that is, a semantic tree of the source code.
 - Trace the generated tree to match the nodes with the semantics of the target machine. If the matching process succeeds, it produces an assembly version of the source programs.

- Assemble and Link

The generated assembly code is then fed to a GNU Assembler, commonly known as GAS, and then to the linker to create the executable file.

3.3.2.3 Why VP?

The grounds for choosing VP as a base language over the alternatives of Java, C and FORTRAN are as follows:

- VP supports features that implicitly express data parallelism such as array features.
- The target language's front-end compiler is already designed to support SIMD extensions and a flexible degree of parallelism.

- Java was not targeted due to the difficulty of efficiently transmitting data parallel operations via its intermediate code to a just in time code generator [7].
- C was not considered because it considers array names as pointers. It also allows arithmetic operations on pointers, and consequently any operation in C involves array names within indices will be carried out on the pointers, which usually point to the first element in the involved arrays, not on data[35].
- The FORTRAN language has been targeted by one of our group to port it also on the Cell processor.

3.4 Assembly Language Directives

The GNU Assembler, which is frequently abbreviated to “GAS” or “as”, was used in the PowerPC machine descriptions. The GAS assembler with the AT&T syntax was used because it is supported on Linux machines. GAS, as claimed by some users [144], was not an easy tool to use. It also lacks documentation.

We present here are some of the GAS directives [145] that were used in implementation of the original PowerPC machine description and the extended version.

- `.data subsection`: Informs the assembler to put the following statements onto the end of the given data subsection.
- `.text subsection`: notifies the assembler to put the following statements onto the end of the given test subsection.
- `.extern`: for compatibility with other assemblers.
- `.align expr`: Pads a memory location to a given storage boundary. For example, `align 3` skips to the first memory location that is a multiple of $2^3 = 8$.
- Labels: A symbol that is followed by a colon “:” can be used to reference a memory location [146]. It can be also used as an instruction operand. Users are not allowed to use one symbol to represent two memory locations.
- Macros: `.macro` and `.endm` are the two commands that can be used to define macros which generate assembly code.

4 Related work

The advent of heterogeneous architectures in the mainstream industry has a significant influence on mainstream software as they require proper, simply-used and up to date tools for developing parallel and concurrent programs [13, 15, 147]. In recent years, substantial effort has been put on designing and developing parallel tools for programming general purpose application on heterogeneous architectures such as multi-core General Purpose Units (CPUs) and Graphics Processing Units (GPUs) [46, 5, 19, 62, 34, 44, 25, 38, 33, 79, 30, 8, 52, 148, 149, 60, 50]. CPUs and GPUs are distinguished from each other in the number of cores and their speed, but the parallel software tools that these two heterogeneous architectures require are conceptually similar. Heterogeneous architectures basically require software components that can make use of different processing elements based on their capabilities [15, 13]. Multi-core CPUs consist of tens of cores and can handle only few tasks yet very quickly. In contrast, modern GPUs usually contain hundreds of cores that can be used to process hundreds or even thousands of tasks concurrently with reasonable speed, and their performances have been increasing rapidly in recent years [150].

This chapter presents a number of programming paradigms that have been recently developed to simplify parallel computation on heterogeneous shared-memory architectures. It first introduces two parallel programming models; CUDA and OpenCL, that have been developed for programming special purpose machines and modified lately to make use of GPUs for general purpose computing [95, 151, 148]. After that, it introduces two other parallel programming models; OpenMP and Intel TBB. These models have already being in use on other architectures but have been recently modified for programming the Cell processor. It also introduces other models that have been designed specifically for the Cell processor such as Offload C++, CellVM, Hera-JVM [38, 29, 48, 49].

4.1 Compute Unified Device Architecture (CUDA)

The GPUs computation power have encouraged software developers to introduce parallel programming tools that allow using GPUs for general purpose computing [13, 152, 51,

15]. CUDA is one of these parallel programming models that aims to simplify programming GPUs for general purpose applications in addition to performing traditional graphic computation. It was developed by Nvidia, and it supports a number of programming languages such as C/C++, Fortran and OpenCL [71]. This programming model was specifically designed to run on Nvidia's graphics cards. It considers an ordinary CPU as a host and a graphics card as parallel platform that has a large number of arithmetic execution units. Provided that, a CUDA program is normally constructed of one host process and one or more accelerate processes. The host process should handle the parts of the program that cannot be parallelized, while the accelerate processes perform the parallel computation [152]. CUDA programming is basically based on three main concepts: Threads, shared memory and kernels.

- Threads

The units of work in CUDA are organized into a hierarchical form. The bottom level of the hierarchy is called Threads. A thread is the smallest unit of parallelism in CUDA. The second level is called Warp. It is a group of consecutive threads that execute in parallel. The third level is called blocks, and each block embraces multiple warps. Threads of one block can synchronize and quickly communicate between each other. The top level of the thread hierarchy is called Grid[152]. It is a group of blocks. Once a grid is launched, all threads of that grid must complete execution before executing any other threads. Blocks within one grid cannot synchronize with each other.

- Memory

Memory is also organized into a hierarchical form. The main memory is considered as a top level (global) resource [71]. CUDA threads can read and write to the global memory. The mid level of the memory hierarchy is called shared memory. Each accelerator is designated a small part of the shared memory. The shared memory of a given accelerator is subsequently divided on a block by block basis to be used as a working space for threads within a block. By designating a shared working space for each block, the communication between threads within one block becomes easy and quick.

- Kernels

Kernels are data-parallel functions or subroutines that are used to differentiate parallel code from sequential code. A kernel function generates a grid of thread blocks.

4.1.1 Compilation Process

The source code of CUDA applications should be saved in “*.cu” files. A single CUDA program supposes to include host code and kernel functions. The host code is ordinary C/C++ code that runs on the host processor while the kernel functions run on the GPU. Once a source file is supplied to the CUDA compiler system, the compilation process includes the following steps:

1. Splitting the host code from the kernel function and keeping it in file with an extension “*.cup”. This process is called a preprocessed source file.
2. Separating the preprocessed code into two files:
 - The host code is kept in files with conventional C/C++ extensions such as “*.c”, “*.cc” or “*.cpp”
 - The GPUs intermediate code is kept in a “*.gpu” file.
3. Compiling the host code by a host machine’s C/C++ standard compiler
4. Compiling the kernel function using an NVIDIA compiler device.
5. Linking the compiled code with CUDA run-time libraries.

4.1.2 Program Execution

A typical CUDA program execution is as follows:

- Executing the host code
- If a kernel function is encountered, the execution of that kernel function is then dispatched to the accelerate devices by the host processor using a special procedure calling.
- The kernel generates a large number of threads that run on GPU. The generated threads are grouped in grids.
- Once all the generated threads complete their execution, the analogous grid terminates, and the execution of the host code resumes until another kernel is encountered.

CUDA is recommended for data-intensive applications in which the computation can be split into hundreds or thousands of threads [152]. Programming specialized processors, such as GPU, is usually more complicated than programming CPUs.

4.2 Open Computing Language (Open CL)

OpenCL is a heterogeneous computing environment that is based on the C99 standard of the C language [52, 151]. The first stable version, OpenCL 1.1, was released in 2010. It was developed by the Khronos group and other industry companies and institutions[53]. It was designed for developing general purpose applications that run across heterogeneous platforms containing CPU, DSP, GPUs, and other architectures [151, 148, 53, 32]. An OpenCL program presumes that a heterogeneous platform is built of one host processor and one or more accelerator devices. It also presumes that data to be processed in parallel is indexed (represented) in an N-dimensional space where $N=1,2$, or 3. The index space is essential to parallelise a problem, for example, parallelising a vector of data could be handled by one ($N=1$) dimensional space while processing 2D arrays, such as an image, would be efficiently carried out on two ($N=2$) dimensional space [52].

4.2.1 Program Structure

An OpenCL program is a collection of host code and OpenCL code [151]. The host code, which runs on the host machine, is responsible for:

- Initialising accelerator devices
- Dispatching kernels to the accelerator devices via queues.
- Transferring data
- Synchronising data

The OpenCL code is a collection of kernels that are executed on the accelerator devices. A kernel implementation, which is similar to a C function, is defined using the keyword `kernel` and it is the basic unit of computation. Each kernel execution is called a work-item, and work-items can be grouped into what is called a workgroup [151]. Work-items in one workgroup are executed on one accelerator device and have shared memory space.

4.2.2 Memory Structure

The memory hierarchy is divided into three levels. The top level is called a private memory; it is a designated space for a work-item. The second level is called local memory, and it is allocated for a workgroup and can be shared among the kernels in the workgroup

[151]. The third level in the memory hierarchy is called global memory. It refers to an accelerator device’s memory. A global memory is visible to all workgroups run on the device it belongs to. The bottom level is called the host memory, and it refers to the memory on the CPU. The host memory is visible to all other devices [151]. To distinguish the three levels of memory spaces, OpenCL offers three address qualifiers: `__global`, `__local` or `__private`. It offers also the “`__constant`” qualifier to describe read-only variables [52, 32].

4.2.3 Data Type

The OpenCL language dropped some of the C99 features such as variable length arrays, function pointers, recursion and bit fields, but it supports other features that C99 did not support such as vector data types and vector operations [52]. Vector types can be defined by combining primitive data type names, such as `int` and `float`, with a value n where n specifies the number of elements per vector. For example, if the vector size is 128-bit, then a vector of 4 characters would be defined as `char4` and a vector of 4 floating points would be `float4`. The individual elements can be accessed using the vector variable name plus a dot (`.`) plus an index. There are two ways to index (address) a particular element in a vector data type: character set or numeric index. Using characters, the form is `vectorName.xyzw`. For example,

```
double2 v;
v.x = 2.1f;
v.y = 5.2f;
v.z = 8.6f; //Invalid: vector v has only 2 elements
```

Individual elements can also be accessed using numeric indices. For example, the following code segment

```
float16 fvec;
fvec.s0 = 12.1f; // 1st element
fvec.s1 = 24.0f;
...
fvec.sa = 10.3f; // 10th element
fvec.sb = 11.5f;
fvec.sf = 16.2f; // 16th element
```

declares variable *fvec* of 16 floating-point elements, and then initialises one element at a time. In OpenCL, like in Vector Pascal, operations can be performed on vectors or a vector and scalar. For instance,

```
int16 v1,v2;
int scalar;
v2 = v1*scalar;
```

is equivalent to

```
v2.s0 = v1.s0 * scalar;
v2.s1 = v1.s1 * scalar;
...
v2.sf = v1.sf * scalar;
```

4.2.4 Built-in Functions

The OpenCL language provides a set of specialized built-in functions for query operations, math operations, work-item operations, image access and synchronization [52]. Figure 4.1 shows common functions:

4.2.5 Program Execution

To illustrate an OpenCL program execution, we first present sequential and parallel versions of a code for doubling elements of a vector and then explain the execution process. Figure 4.2 shows the scalar implementation in C for doubling n elements of vector X and storing the result in vector Y . Figure 4.3 shows a parallel OpenCL code for the same purpose.

The C function in Figure 4.2 is a sequential implementation that iterates using an *for* loop structure through the n elements in the received vector X , doubles one element at a time and saves the result in the vector Y .

The code given in Figure 4.3 is a shorter OpenCL implementation of a kernel. It does not include, for the sake of simplicity, details on how to set up a parallel environment such as querying for platform information, creating a context, creating memory object...etc.

4 Related work

```
// Returns an unique global ID for a work-item in the dimension space idx
    size_t get_global_id(idx);

// Returns the number of work-items (kernels) in the dimension space idx
    size_t get_global_size(idx);

// Sets barrier to block a work-item until others (same group) have executed
    barrier(_flag);

// Takes image and coordinate and returns a vector of color values
    float 4 read_imagef(image, coord);

// Write color value to coordinate (coord.x,coord.y)
    void write_imagef(image,coord,float color);

// Returns width of an image in pixels
    int get_image_width(image);
```

Figure 4.1: OpenCL Functions

```
void vecDouble(const int n,float *X, float *Y){
for ( int i = 0 ; i < n ; i++)
    Y[i] = X[i] + X[i];
}
```

Figure 4.2: Scalar C Function

```
kernel void vecDouble(global float *X , global float *Y) {
int id = get_global_id(0);

    Y[id] = X[id]+ X[id]; // run over n work-items
}
```

Figure 4.3: Parallel OpenCL Kernel

The function header defines it as a kernel which implies that this code (function) will be dispatched on the available accelerators but works on different data. The available accelerators needed for processing all data should be predefined. The kernel receives two global pointers of type float: the pointer X refers to the input data and Y refers to memory location on the CPU into which the result is stored. When this kernel is executed on an accelerator device, the first line in the kernel body requests a global ID to index data that will be processed by that accelerator. The second line then uses the ID to perform the requested operation on the id^{th} element in the vector X and store the result in the id^{th} element in the vector Y .

OpenCL supports a number of architectures such as AMD, IBM, Intel and Nvidia. It supports data and task parallel computing. A task is executed as a single kernel [53]. It also supports run-time compilation which allows applications to utilize different compute devices at run time [52]. OpenCL, however, requires users to learn how to develop parallel applications that are efficient and can perform well on different heterogeneous platforms [53]. To overcome the programming difficulties that are still associated with OpenCL, a new source-to-source tool called Swan has been recently introduced by Imperial College London. Swan was designed to port CUDA to OpenCL by converting existing CUDA code to OpenCL code [50]. This tool takes advantage of the OpenCL support for broad architectures by helping to run existing CUDA applications on architectures other than Nvidia.

4.3 Programming the Cell BE Architecture

This section presents a number of parallel programming paradigms that have been developed recently for programming the Cell processor. Some of these models have been designed to exploit data-level parallelism. Data-parallel based models, such as OpenMP, SieveC++ and Offload, basically focus on data parallel code such as loop structures. Other models have been developed for exploiting thread-level (task-level) parallelism. Task-based models, such as CellVM and Hera-JVM, have been designed for managing multi-threaded applications.

In the following discussion, we start with OpenMP as a widely used API for programming the Cell processor and other architectures. After that we talk about two thread-level based models that were introduced two years ago: Hera-JVM and CellVM. We shall talk also about SieveC++ as source-to source compilers, and finally look at Offload C++ and Threading Building Blocks (TBB) tools that have been developed very recently to write programs for the Cell processor.

4.3.1 Open Multi-Processing (OpenMP)

OpenMP is an API that has been designed to support parallel programming of general purpose applications on architectures ranging from personal computers to supercomputers [30, 27]. It supports C/C++ and Fortran Languages, but the discussion here is based on C++ code. Explicit-based programming models usually require users to set up resources for parallel environment and to instruct the compiler on the parts of code to parallelize. OpenMP offers programmers the tools to explicitly manage multi-threaded parallelism on shared-memory architectures [30]. These tools include a set of environment variables, compiler directives, and library routines:

- Environment variables

To control the execution of parallel code, OpenMP provides the following four environment variables:

- `OMP_NUM_THREADS`: sets the maximum number of threads.
- `OMP_SCHEDULE`: determines how iterations of a loop are scheduled (static, dynamic ...etc).
- `OMP_DYNAMIC`: sets/resets dynamic adjustment of the number of threads available for execution parallel sections.
- `OMP_NESTED`: nests one parallel section into another parallel one.

- Data environment management

By default all variables within OpenMP code are visible to all threads [30]. Users, however, can manage variables in several ways using the following data constructs,

- *Shared*: It implies that data within a parallel section is shared among threads.
- *Private*: It means that data within a parallel section is private to each thread. That is, each thread has a local copy and uses it as a temporary variable.
- *Default*: It allows specifying the default data scoping within a parallel region.
- *flush*: It can be used to restore the value of a given variable from register into the memory for using it outside the parallel region.

- Preprocessor Directives

OpenMP offers several types of directives. Different languages use different pre-processor directives. For example, in C/C++ a directive must begin with the keyword (*#pragma omp*), while in Fortran directives must start with, *!\$OMP C\$OMP*, or **\$OMP* depending on the version of Fortran [27, 30]. We use C++ code to illustrate the use of the following directives:

– *Parallel* Constructs

C++ provides three different OpenMP constructs to define region. They are as follows:

- * A (*for*) directive: it splits up loop iterations on threads, but it does not create threads. It is practical for implementing data-parallelism.
- * A *parallel* directive: it splits the current thread into a new group of threads which stay active until all the created threads are completed and merge back into one thread.
- * A (*parallel for*) directive: it is a two command directive: *parallel* and *for*. The first part or command, as mentioned above, creates a new group of threads and the second command splits that group to work on different parts of the data.

– *Sections* Constructs

It is used to run consecutive code blocks on different threads. It works well for task parallelism.

– *Master* Construct

It specifies a code block that will be executed by the master thread only.

– *Single* construct:

It can be used to determine a code segment that will be executed by one thread. A barrier is implied at the end because other threads can skip that thread and wait at the barrier.

• Synchronization constructs

- *critical section*: executes the defined section by all threads but only one at a time. It is usually used to protect shared data from race condition.
- *ordered*: the enclosed code block is executed in order.

- *Barrier*: enforce each thread of a given section to wait until all threads of that section reach this point.
- *nowait*: permits completed thread to proceed.
- Scheduling clauses

Schedule clauses are very useful for *for-loop* constructs because it divides work of loop iterations among different threads. For this purpose, OpenMP offers the following three scheduling mechanisms:

- *Static*: Iterations are divided among threads equally before the execution start.
 - *dynamic*: iterations are divided into small groups using a small number of threads. Once a thread finishes its task, it requests another chunk from the ones that are left.
 - *Guided*: chunks of consecutive iterations are dynamically allocated to each thread. With this mechanism, the chunk size is decreased exponentially each time. This approach suits loops that do not have constant cost per iteration such as in Mandelbrot applications.
- Library routines

They could be used to attain information on threads such as thread identifiers or the total number of threads. . . etc.

4.3.1.1 OpenMP Program Execution

The following is a short description of how a program with OpenMP directives is executed:

1. The program starts as a single thread. It is called a master thread and its ID is “0”.
2. The master thread executes sequentially.
3. During the execution of the master thread, if it encounters a parallel region, it then
 - a) Creates a number of threads, and each thread has an integer “ID” but not “0”.
 - b) Forks the region to be parallelized among the created threads.

4 Related work

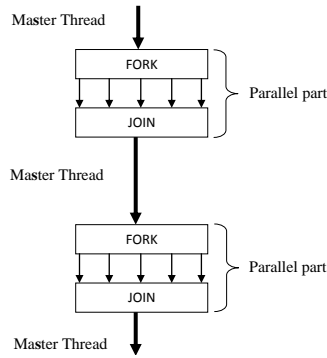


Figure 4.4: OpenMP Program Execution

- c) Once the execution of the parallelized region is finished, the threads then are synchronized, terminated, and joined back into the main thread. See Figure 4.4.

4. Resume the master thread execution
5. Go to step 3 if it encounters another parallel region.

4.3.1.2 OpenMP on Cell

OpenMP provides preprocessor directives in Fortran and C/C++ to control compilation for the Cell processor. The OpenMP compiler for Cell is a single-source compiler and handles data partitioning, communication and synchronization automatically [79]. The compilation process is carried out using two parts: a source-to-source front-end compiler or translator and existing PPE and SPE compilers. Given a C program, the OpenMP compiler first generates dual source code: PPE source code and SPE source code. The PPE code includes all sequential code which is handled by the master thread. The master code is compiled by the PPE compiler. The other source regions, which are supposed to be defined as parallel, are translated into SPE code which is compiled using an SPE compiler [27]. To illustrate the compilation process, we present the following simple example:

```
#pragma omp parallel for
for (int i = 0; i < N ; i++)

    Y[i] = X[i] + X[i];
```

The code segment shown above is C source code that defines a parallel *for* loop. The loop goes over vector *X* of length *N*, doubles each element and stores the results in vector *Y*.

The first line of code defines a *parallel for* directive which generally results in creating a new group of threads to execute the corresponding task. When the compiler encounters this directive, it offloads the *omp* section on the available SPEs by creating a new function and cloning the code on the SPEs. At the run-time, each SPE function uses DMA transfers to grab blocks of data for computation, and when the task of each cloned SPE function (SPE thread) is finished, each SPE thread then acknowledges the PPE via mailbox or signals with the completion.

Generally, OpenMP assists in relaxing some of the complexities, such as data partitioning and communication, that are associated with the development of parallel applications. These features as well as being a single-source compiler have made it a widely used API for multi-core architectures. However, using OpenMP directives may result in performance degradation if the selected region is not worth parallelizing. The degradation in performance is due to the gained speed up or improvement not paying off overheads of thread creation, shared-memory activities and synchronization. Thus, the development of OpenMP program often requires some skill and effort to get the best performance. It also does not support parallel I/O operations such as write/read from a shared file.

4.3.2 Sieve C++

Sieve C++ is a parallel programming system that was released by Codeplay in 2006 [18]. It is another programming model that aims to simplify developing parallel applications for multi-core architectures. The main concept of Sieve C++ is to split sieve code into three components: reading, computing and writing [22]. This strategy allows parallelising the computation part safely as memory access operations are separated from the computation. The Sieve system is an extension to a C++ compiler that includes a run-time system for processing management [18]. The system is basically a source-to-source compiler that takes in C++ source code that is wrapped inside a sieve block, distributes work across multiple processors and most importantly instructs the compiler to delay side-effects within a sieve block until the end.

4.3.2.1 SieveC++ Code Structure

Sieve uses a wrapping technique to enfold code inside blocks. A sieve block must be annotated with the keyword `sieve`, and it could be a function or a region of code that is enclosed within braces. To define a function as a sieve block, the keyword `sieve` must be placed immediately before the semi-colon symbol (;) which marks the end of a statement

```
returnType functionName ( arguments list ) sieve;
```

Figure 4.5: Sieve Function Prototype

```
sieve { // The start of a sieve block
    ...
    ...
} // The end of a sieve block
```

Figure 4.6: Sieve Block

in C. The general forms of a sieve function and a sieve block are shown in Figures 4.5 and 4.6 respectively.

In Sieve C++ system, a sieve block could be split into multiple processes if a dependency exists between variables inside a sieve block. If local variables in a sieve block have dependencies, the sieve compiler will then implicitly split code into chunks which are processed in sequential fashion [18]. The sieve compiler, however, could be explicitly instructed by using a *splitthere* statement to split up code within a sieve block into independent processes [38].

The sieve system provides program developers a number of special library classes. We introduce here two examples of these classes: `IntSum` and `IntIterator`. The `IntSum` is an accumulator class. Accumulator classes can be used to split an operation on multiple processors [18]. The `IntSum` class, for example, suggests that the sum operation is carried out in parallel, and thus the Sieve compiler splits any variable of type `IntSum` into multiple variables each evaluated on different processors. The `IntIterator` class can be used to split an iterator. The `IntIterator` class splits the iterator control object or “index” into multiple indices and sent them to different processors. These classes could be also applied to other data types.

4.3.2.2 Sieve Side-effect Rule

Side-effects in a computing environment mean modifying or updating variables or states of shared types such as global variables, static variables and pointers [18]. It is a fundamental concept to handle dependency that often appears with automatic parallelisation. The SieveC++ compiler is designed to delay all side-effects within a sieve block until the

```

void vecDouble(const int n, float *X, float *Y) {
    sieve {
        for (int i=0; i< n; i++)
            Y[i] = X[i] + X[i];
    }
    // The assignment to vector Y delayed till here
    ....
}

```

Figure 4.7: SieveC++ Implementation

end of the block. To illustrate how the dependency can be solved by the side-affects deferring technique, we present the same example given in Figure 4.2 to double elements of vector X and store the result in vector Y , but this time the loop is defined as a sieve block, as shown in Figure 4.7, to be parallelised by the SieveC++ compiler.

The *for* loop structure in Figure 4.7 is declared as a sieve block, and therefore the addition operation will be carried out in parallel on separate processors and assign the result in vector Y . Under the sieve system, the compiler will delay the assignment to vector Y till the end of the loop to avoid any dependencies that might appear as a result of the pointer X and Y point to memory spaces that overlap.

This side-effect approach gives the compiler the flexibility to reorder the execution of the statements within the block and also eases the task of the compiler to partition the sieve code [18, 38]. As a result of that, the compiler is required to do dependence analysis only on local memory locations. This approach also provides more memory space to be used as data outside a sieve block is separated from the inside data and will not be processed until the end of the sieve.

4.3.2.3 Compilation Process

The Sieve C++ system handles standard code differently from sieve code. Code that is outside sieve blocks is compiled as a normal sequential code while the compilation process of sieve code goes through several steps:

- The compiler starts with dependency analysis on each sieve block to determine the split points where there is no dependency.
- If the compiler comes across any dependencies within a sieve block, it will then report to the user where and on which variable a dependency exists [18]. Such

```

int Reduction(int *X, int size) {
    int Result;
    sieve {
        IntSum subTotal(0) splits Result;
        IntIterator index;
        for(index=0;index<size;index++)
            SubTotal += X[index];
    }
    return Result;
}

```

Figure 4.8: Sieve C++ Code

information is very important for programmers in order to do the proper changes for a better parallelization.

- Splits up the code within a sieve implicitly into independent processes.
- Once the splitting is done, the compiler then diffuses in parallel the processes on the available processors.

4.3.2.4 Sieve Block Execution

On entering a sieve block, the sieve run-time system starts distributing processes among multiple cores, and once a process execution reaches an exit point, it returns to the run-time system to decide which process to run next [38]. To illustrate the use of iterator and accumulator classes as well as the execution process, we present as shown in Figure 4.8 a simple reduction addition function.

The first two lines in Figure 4.8 are standard C code to define a function heading and to declare an integer variable called `Result`. Moving inside the sieve block, the first statement declares the `subTotal` variable as an object of type `IntSum` and initialised with 0. This implies that each process or thread will maintain in its `subTotal` variable the sum of some elements in vector `X`. At the end of the execution of the sieve block, the sums in the `subTotal` variables are assembled in the `Result` variable. The elements of vector `X` that each process is supposed to sum are determined by the next statement in the block which declares the variable `index` as instance of class `IntIterator`. This results in splitting up the `index` variable on the available processes. If p processors were used, then the number of elements ($elem$) that each processor sums together is $elem = \frac{index}{p}$, and

a processor N where $0 \leq N < p0$ start iterating from $N * elem$ and stop when reaching $(N + 1) * elem$.

4.3.2.5 Sieve C++ on Cell

The SieveC++ system's main concept, which is based on separating memory access from computation, suits the Cell BE architecture because it has multiple levels of memory structure [22]. The Sieve-Cell system; which was initiated as an experimental attempt, works on the Cell as follows:

- A sieve block can be used to determine the part of code that can be executed on the SPEs
- Variables outside a sieve block are located in the PPE memory
- Local sieve block variables are located in the SPEs local storage.
- Reading global memory within a sieve block triggers a DMA transfer to move data into an SPE local storage.
- Writing global memory by a sieve block are kept on SPEs local storage to avoid any side effects. They are delayed to be flushed into the memory at the end of the block.

With the Sieve-Cell system, programmers are required to write a single source code that would run on both the PPE and SPE . The compiler processes a Sieve code and produces multiple ANSI C files. These files can then be compiled for the PPE and SPE using a third party C++ compiler and linked using the Sieve run-time libraries to produce an executable file that run on the Cell. The following steps describe how a sieve code is executed:

- The PPE loads each SPE with a program. This program includes a loop that runs continually and code for handling sieve code (aka work unit) execution [22].
- The SPEs uses DMA transfers to request work units from the PPE through their outbound interrupt mailbox.
- The PPE responds through the SPEs inbound mailboxes by sending the pointers of the work unit which are needed to be executed on the SPEs.
- Each SPE then uses the received pointer to fetch the corresponding work unit.
- The PPE instructs the SPEs to start the execution of the work units and waits for them to finish.

```
offload returnType functionName ( arguments list );
```

Figure 4.9: Offload Function Prototype

- Side-effects are handled by a specific SPE function. In case an SPE's local storage reaches a certain limit, that function then uses the PPE memory as a temporary space [22].
- When the SPEs acknowledge the work completion, the PPE then resumes either to issue new work units to the SPEs or to continue the execution of the standard code.
- On the completion of the SPEs work, the PPE must ensure to flush the delayed side-effects in the same order.

4.3.3 Offload C++

Offload C++ was introduced in October 2009 as an extension to C++ for parallel programming, and it was developed also by Codeplay [60, 149]. It is based on the offload technology which focuses on extracting parts of a master code to run on accelerator cores rather than focusing on parallelizing code [149, 29]. The Offload model was designed for programming heterogeneous multiple core processors in C++, and the first version was developed specifically for writing game applications on PlayStation3. In 2010, Codeplay released the second version of the Offload C++ suite for programming the Cell BE processor under Linux [149, 60]. The Offload suite includes a multi-core run-time library, compilers and a debugger.

The Offload model, like Sieve C++, is based on wrapping techniques to determine the code to be offloaded. An offload block must be annotated with the keyword *offload*, and it could be a function or a region of code that is enclosed within braces. To define a function as an offload block, the keyword *offload* must be placed before the function name; see Figure 4.9.

The Offload compiler offloads annotated parts in large C++ programs to the Cell's SPEs using threads [29]. Besides, any function that is called from within an offload block is complied for the SPEs, and data defined inside an offload block are located in the SPEs local storage. However, data movement between the PPE main memory and LSs is automatically handled by the compiler. Code inside an offload block runs in sequential fashion as a thread on an SPE, and thus multiple offload threads can be executed in parallel. In contrast, master code that is not wrapped in an offload construct runs on the PPE. A C++

```

int *X;                                // Outer pointer
int main() {
    offload {
        int *Y;                        // Local (inner) pointer
        int Z = *X;                   // Requests DMA
        X = Y;                        // Error Y is not outer
        foo (X);                      // Valid function call
        foo(&Z);                      // Error \&Z is not outer
    }
}

```

Figure 4.10: Offload C++ Code

program that includes offload constructs is compiled to intermediate C code for the PPE and SPEs. The code is then compiled using GCC compilers for the PPE and SPE. In what follows, we briefly describe the main design issues in Offload as: Offload scopes, Dereferencing pointers and Duplication technique.

Offload scopes refer to offload blocks and offload functions [29]. Non-offload variables, such as global variables, can be accessed by offload scopes using replicating techniques. For example, Offload allows an offload scope to use global variables by copying the global variables to local variables inside the offload thread using the same names. The copied versions then can be used by an offload thread to refer to the global variables. The second important issue is how to distinguish between host memory pointers and local memory pointers.

Offload provides the *outer* qualifier to distinguish between a host memory pointer and a local memory pointer. The *outer* qualifier can be used to refer to host memory pointer. A pointer that is not inside an offload block is considered *outer* by default, and pointers to local memory are considered as non-outer (inner) pointers. Thus assigning outer pointer to inner pointer and the other way around are both invalid. In the implementation of the Offload on the Cell processor, *outer* pointers cannot point to the SPEs local storage, and *inner* pointers cannot refer to the PPE memory. Nevertheless, if an offload scope dereferences an outer pointer (host memory), this then is solved by transferring data from host to local memory. For instance, on the Cell processor, the data movement is handled automatically by the compiler using DMA. The following example gives an idea on offload scopes and outer and non-outer pointers:

The code given in Figure 4.10 includes one global pointer *X* which is by default an outer pointer and resides in the PPE memory. The first statement in the offload block declares one local (inner) pointer *Y* of type integer that points to local memory location. The next statement also declares a local integer variable *Z*. In the same statement an outer pointer

was used to assign the value in the PPE memory location X into the local variable Z . Now, Z was declared inside an offload scope which implies that Z resides in the local memory, and therefore data should be transferred from the PPE into the SPEs local storage using DMA. This example also shows two instances of invalid assignments in which the code tries to assign inner pointer to outer pointer or pass the inner pointer to another function.

The third technique in Offload is the Call-graph duplication [60]. It is used to provide multiple compilation units which are needed sometimes when non-offload functions are involved. If a program has only offload functions the compiler can easily compile them for the SPEs. However, if a non-offload function is called from an offload scope, the compiler is then required to overload the non-offload function. The new version of the overloaded function should be identical except the new version has the *offload* qualifier.

Offload shows some performance improvement on the Cell accelerator cores, yet it does not completely hide the underlying details of the architecture. The duplication technique may produce a series problem as the Cell accelerators have very small storage space; for instance, if a standard function is called several times by offload scopes, this approach then may dramatically increase the code size [60] .

4.3.4 Hera-JVM

Hera-JVM is a run-time system that supports migration of Java application threads between heterogeneous multi-core architectures [133]. It does not focus on parallelizing code, like the proposed VSM, instead it focuses on managing threads between different core types by looking at the heterogeneous cores as a homogeneous multi-threaded virtual machine [49]. Hera-JVM was implemented for the Cell processor and was presented as a doctoral thesis work at Glasgow university in 2010.

Hera-JVM is built upon a Java Research Virtual Machine (Jikes RVM) which supports PowerPC architectures and allows Hera-JVM to run on the PPE without any modifications [133]. The initiative of Hera-JVM's design is to hide some aspects of the processor's heterogeneity and enable Java programmers to use heterogeneous cores without the need for deep familiarity with the processor's design [49]. Its run-time system depends basically on annotations to gather information on code behavior, and then uses this information to map Java application threads to the proper underlying cores. To trace a program's behavior, Hera-JVM provides a number of tagging mechanisms and behavior characteristics.

It suggested three different tagging mechanisms to describe expected behaviors of source code and to provide the information about the code behavior. These three mechanisms are: explicit annotations, source code analysis tools and run-time monitoring. An explicit

code annotation is the basic tagging approach that is being supported by a number of programming languages [49]. Source code analysis tools, on the other hand, are automatic approaches to track an application's behaviour and tag the proper code segments [133]. The third tagging mechanism is a dynamic approach to monitor code behaviour from different perspectives at run-time.

It also provides three mechanisms to perceive the source code behaviors at different stages: processing behaviour, thread communication behaviour and execution behaviour. The first class provides hints for the Hera-JVM run-time system to make an efficient use of similar processing units based on the capabilities of the similar units on different core types. For example, the performance of the floating-point unit on the Cell processor varies between the PPE and the SPEs. Hera-JVM offers the following tags to characterise the required processing for the main functional units [133]:

- *IntegerCode*: mark code segments that require intensive use of fixed-point unit.
- *FloatingPointCode*: mark code segments that require using floating-point unit.
- *DataAccessCode*: mark code segments that require intensive memory access.

However, if these hints are not significant when the performances of the processing units on the different core types are the same, the Hera-JVM run-time system then omits these hints.

Inter-thread communication is the other aspect that Hera-JVM focuses on. Threads of the same process often work on shared data and require communication with each other, and therefore the information on shared data is important for efficient communication between threads. To achieve that Hera-JVM looks at threads that frequently communicate with each other and then groups them together using the following tag [133]:

@ThreadTeam(name="<name of team>")

Hera_JVM also looks at data locality, distance between the cores accessing the data and the possibility of using efficient communication mechanisms such as scratchpad memory transfers.

The third activity that Hera-JVM traces is the behaviour of program execution. Knowing the expected behaviour of an executing thread helps the run-time system to choose the proper core type on which the thread should be executed. The following five tags are offered by Hera-JVM to characterise program execution behaviour:

- `@SequentialAccessBehaviour`: Tags code that requires access to consecutive memory locations such as arrays.
- `@RandomAccessBehaviour`: To tag code that requires non-sequential memory access.
- `@LargeWorkingSet`: To specify code that is expected to work on a considerable chunk of data and also specify the expected size of data. This information assists the run-time system to determine the proper core that has enough cache [133].
- `@IoAccessBehaviour`: To determine I/O code segments.
- `@ExceptionsLikely`: Tags code that is expected to generate many exceptions. This tag is useful in choosing the core that is less costly than the other cores in handling exceptions.

To determine the influence of each behaviour on a program's performance, each behaviour characteristic should be associated with its cost on the different core types. The cost information on the different type of cores can then help Hera-JVM run-time system to automatically determine the total cost of a given behaviour on the available heterogeneous resources and choose the proper core type that would be efficient to execute a given code.

The Hera-JVM run-time system relies on behaviour characteristic annotations to gather information on a program's execution, and it then uses this information to migrate threads to the proper underlying cores. This means that the Hera-JVM approach does not consider parallelizing code or data, but it provides techniques for managing multi-thread execution on heterogeneous architecture [49]. The Hera-JVM run-time system, unlike VSM, hides only some aspects of the processor's heterogeneity and still requires programmers interference to add annotations.

4.3.5 CellVM

CellVM is a Java virtual machine that can also be used to offload Java threads on a heterogeneous multi-core architecture such as Cell. It aims to exploit the performance potential offered by the Cell processor and to improve programmers' productivity. CellVM basically emulates, similar to Hera-JVM, a homogeneous shared-memory multi-core machine to hide the heterogeneity of the underlying hardware of Cell and to offer a high level of abstraction for offloading individual Java threads on the SPEs [48]. It has been developed to overcome the barriers of the standard Java VM's to incorporate the SPEs for executing Java instructions. CellVM, which was first introduced in 2008, is an extension of a

compact Java VM called JamVM. The main feature of JamVM is its size. Compared with most other JVM's, JamVM is very small which makes CellVM suit the challenges faced by the limited space of the SPEs' local storage [48]. CellVM has been implemented as a prototype system that can be used to execute in parallel one thread per SPE [48]. This prototype system consists of two collaborate Java VM interpreters: ShellVM and CoreVM [48].

ShellVM runs on the PPE to maintain the overall control of the machine resources. It is responsible for handling and executing built-in routines, such as "Math" routines. Built-in routines should not be executed on the SPE because first they usually require a relatively large memory space while the SPEs local storage is very limited [48]. Secondly, built-in routines often uses Java heap which resides in the main memory, and thus executing built-in routines mostly likely will require access to Java heap space and this process will be very costly. ShellVM is also responsible for code-preparation to save switching between PPE and SPEs. It prepares the bytecode of any routine before calling that routine [48]. The other task of the ShellVM is resolving references that are not known at compile time. The preparation process, which can be handled in the preparation stage, deals only with dynamic references that are not altered during the entire execution of application. The current implementation of CellVM has solved this issue by adding 8 bytes to each bytecode to store dynamic information. This approach is expected to reduce the number of DMA transfers and the switching between threads.

The other interpreter, CoreVM, can be seen as a virtual Java code execution unit on an SPE. The currently developed CoreVM is expected to execute most Java instructions, but if it finds an instruction that it cannot execute on the SPEs, the CoreVM then hands the instruction to ShellVM [48]. For performance considerations, the CoreVM implementation also excludes a subset of the JamVM functionality, such as managing Java heap and imported functions that are written in other languages. This management process is left to the PPE to handle for the following reasons: First Java heap space, which is often used for allocating new objects such as arrays and sometimes by external functions, resides in main memory, and therefore the cost of accessing the main memory from the PPE is less than accessing it from an SPE which requires using DMA transfers [48]. Secondly, executing imported functions on an SPE needs to offload the function's code on the SPE local storage, and this means reducing the available space for manipulating data on the SPE.

A program execution under the CellVM starts with the ShellVM to set the spaces required for imported functions and data such as Java heap. It then prepares the required information, such as data structures and code, for launching the CoreVM. Once this information is cached in the local stores on the SPEs, the CoreVM then starts executing the code which is expected to continue unless the CoreVM interpreter encounter code, such as

imported functions, that must be executed on the PPE. If CoreVM needs such assistance from the PPE, the procedure to switch the instruction's execution from CellVM to ShellVM is then as follows: First, updating the status of CoreVM on the PPE using a DMA transfer. Once the DMA transfer is completed, the ShellVM then postpones the CoreVM and starts executing the requested instructions. After ShellVM finishes executing the requested instructions, it acknowledges CoreVM with the completion, and CoreVM resumes with execution.

The CellVM model can only be used for Java multithreaded applications because it does not have the capability to divide workload of a single thread. The other constraints of the current implementation is the number of threads that can be executed per SPE. For example, If a multithreaded application has threads more than the available SPEs, the additional threads then are executed on the PPE. This implies that the performance potential of the Cell processor can be reached only if the number of an application's threads is equal to the number of available SPEs. Handling imported functions also degrades the performance of CellVM because they often require switching between CoreVM and ShellVM. The CellVM design also does not completely hide the underlying details of the Cell processor and hence programmers must be very familiar with the Cell processor's architecture in order to exploit its potential performance.

4.3.6 Threading Building Blocks

We introduce here another approach for programming the Cell processor. This approach combines heterogeneous and homogeneous parallel programming models. The first model is the Offload C++, and it has been already introduced in this chapter. The second model is the Intel Threading Building Blocks (TBB). This section first describes the Intel TBB approach and then explains how these two models were integrated to port C++ on the Cell processor.

The Intel TBB is a C++ run-time library that is introduced to simplify development of parallel programs on Intel homogeneous platforms [153]. It provides high-level thread management through the run-time libraries to liberate programmers from managing threads explicitly which often requires thread creation and explicitly mapping the individual subparts of a given problem onto threads efficiently. These activities in TBB are handled by a task-scheduler [153]. The TBB library also provides other low-level features such as locks, atomic operations [20], and it makes use of the template feature in C++ to provide a number of generic parallel constructors such as

`parallel_for` , `parallel_reduce` and `parallel_while`

The TBB task scheduler automatically determines the number of threads and arranges their execution. A thread in TBB must be an initialised object of type *tbb::task_scheduler_init* as shown in the following code:

```
using namespace tbb;
int main( ) {

    task_scheduler_init initThread;
    ...
    return 0;

}
```

The above code declares the object *initThread* without any argument. The default constructor of *tbb::task_scheduler_init* does the initialisation. The constructor can take an optional parameter to determine the number of threads required, but it is recommended not to determine the number of threads. At the end of the program, the destructor should be called automatically to terminate the object *initThread* [153, 20]. However, a thread may have multiple task scheduler objects at a time, and since the overheads of starting up a task scheduler and shutting down are high, it is advisable that the number of created task scheduler objects is kept to the minimum [153].

The *parallel_for* is a template function that can be used to parallelise tasks within a *for* loop structure. It basically chops an iteration space into blocks, and then schedules these blocks to run on different threads [153]. It requires two objects:

- An object of type *range* that determines the iteration space. TBB offers two range types: *blocked_range* for single dimensional spaces and *blocked_range2D* for two dimensional space.
- A *body* object is a form of a function object that includes a modified original serial loop that runs on sub-ranges determined by the range type. The function object should have a copy constructor to copy the code for each thread, and its member function *operator()* is overloaded to accommodate the modified original serial loop.

Figure 4.12 shows a simple program that adds each element in vector *X* and an element in vector *Y* and stores the sum in the corresponding element in vector *Z*. The computation is performed in parallel using the TBB library. The program starts by declaring three vectors; *X*, *Y* and *Z*. The size of these vectors is defined in the first line in the program. It then instantiates a task scheduler object *initThread* which manages the scheduling of tasks on threads (available cores). The third statement in the body of the main function, calls the

```

class vecAdd {
    float *v1,*v2,*v3;
public:
    // constructor copies the arguments into local space
    vecAdd(float *p1,float *p2,float *p3):v1(p1),v2(p2),v3(p3) {}
    // Overloaded operator () to add elements of two vectors
    void operator()( const blocked_range<size_t> &r ) const {

        for(size_t i= r.begin() ; i != r.end() ; i++)
            v3[i] = v1[i] + v2[i]
    }
};

```

Figure 4.11: C++ Class

```

const size_t Size = 10000;
int main() {

    float X[Size], Y[Size], Z[Size];
    task_scheduler_init initThead;
    parallel_for(blocked_range<size_t>(0,Size), vecAdd(X,Y,Z) );
    return 0;
}

```

Figure 4.12: C++ Code Segment Using TBB libraries

parallel_for function which takes two arguments: the first is a range object which determines the iteration space. In the example given in Figure 4.12, the iteration space starts from 0 until Size. The second argument is a function object. It is the constructor of the class *vecAdd* which is defined in Figure 4.11. The constructor stores the received pointers to *X*, *Y* and *Z* as private members. The task of the *parallel_for* is here to create objects of type *vecAdd* and passes on the sub-ranges to each thread. The *vecAdd* objects are then scheduled and execute in parallel[20]. Notice here neither the tasks nor the number of threads were specified; these parameters are set automatically by the task scheduler.

4.3.6.1 TBB on the Cell

Intel TBB primarily does not support heterogeneous architectures such as the Cell BE, but it has been combined with the Codeplay's Offload C++ to allow programs that are parallelised using TBB to run on the Cell's accelerator cores [28]. This work was initiated as an experimental attempt to use TBB on the Cell processor. TBB and Offload are two complementary models. Offload C++ provides, as mentioned before, offload basic construct to wrap regions of code in a given application. The code inside offload blocks is executed on the Cell's accelerators, and any code, which resides outside the offload blocks, is executed on the PPE. Recall also that in Offload C++ data transfers between the host and the accelerators and code duplication are handled automatically while thread management is handled manually. On the other hand, the Intel TBB offers a task scheduler which automatically manages threads. These complementary features were combined to allow code to execute across the Cell's SPEs in a work which was presented as experimental study in 2010 [28].

The combination approach of the Offload C++ and Intel TBB focuses on offloading the parallel loop constructs of TBB in order to port TBB programs on the Cell processor. The implementation, which consists of several template classes, aimed to distribute TBB loop iterations across the PPE and SPEs [28]. The distribution process is carried out by a template function that is injected inside the TBB loop constructs, such as *parallel_for* and *parallel_reduce*. To illustrate this, we briefly describe how the work given in [28] implemented the *parallel_for* construct using Offload C++.

The *parallel_for* construct has two arguments (objects): iteration space (range) and function or code (body) that runs on sub-ranges determined by the first object. The proposed function to be injected inside the *parallel_for* also has the same arguments; range and body. Therefore, by injecting a function inside the *parallel_for* construct, the injected function should be called with the range and the body. On the other side, the responsibilities of the injected function are as follows:

4 Related work

- Get the number of available SPEs; NUM_SPES.
- Get the start and end of the iteration space.
- Determine the block size by equally dividing the iteration space on the NUM_SPES plus one to run on the PPE.
- Iterate on the SPEs to execute NUM_SPES sub-ranges:
 - Compute the beginning and the end of the sub-range, the i^{th} SPE.
 - Call the body from inside an offload block.

Notice here that the second step will trigger the Offload's call-graph duplication because the function (or passed body) is a non-offload function, and because it is called from an offload scope, the compiler in this case will overload the non-offload function on the SPE. The overloaded version is identical to the original code, but it is added the *offload* qualifier.

- Execute one sub-range on the PPE starting with the iteration from the point of the last SPEs .

Thus combining the Intel TBB and Offload C++ as shown above allows parallelising the *parallel_for* construct on the Cell processors.

5 Virtual SIMD Machine

The work here represents the central objective of this dissertation. The basis of this work is a new parallelisation approach to develop a Virtual SIMD Machine (VSM) that can automatically parallelise large data structures on heterogeneous architectures. The novelty in this approach is using a VSM model to completely hide the underlying details of heterogeneous architectures [154]. This approach is based on new parallelisation techniques that imitate a SIMD instruction set using virtual (large) registers and Virtual SIMD Instructions (VSIs). The VSM model is designed to look at machine accelerators as if they are arithmetic units, and when a single virtual instruction is initiated, the actual operation is performed by the accelerators on adjacent parts of the data elements in the virtual registers in parallel.

5.1 Introduction

The VSM model is a register-based model that provides easy access to the target accelerators. The key aspects of VSM design are: its internal (hidden) structure and its interface to users. Its internal structure, which is completely hidden from the users, is primarily built of two co-operative interpreters: master (or host) and accelerator (or slave) interpreters.

The master interpreter schedules micro-tasks and sends messages to the accelerators requesting them to perform an array operation, such as load, store, add ...etc, in parallel. The master interpreter is a collection of stub routines that are run on the host processor, and these routines are responsible for creating and launching threads, data partitioning, communication and coordinating with accelerators for any required synchronization. The stub routines that are responsible for processing data can be seen as virtual SIMD instructions. Most of the VSM instructions were designed as non-blocking operations which means that the master processor can resume executing next instructions once the sent messages have been delivered to the accelerator. This infers that the master processor doesn't have to wait for the accelerator devices to finish the requested operations in order to execute the next instruction. The VSM instruction set also includes a few blocking instructions. The

blocking instructions, such as Store and Reduction, are usually required to send data back from the accelerators to the master's memory space.

The other (or slave) interpreter, on the other hand, is a program that is designed to run constantly on each accelerator in the background. This program is responsible for extracting information from messages sent by the master, any data alignment, acknowledgments and synchronization. An accelerator's program frequently checks if there is any message dispatched by the master or not, and when it receives a message, it decomposes these messages to determine the required operation and then performs the requested operation. The VSM concept fits well with machines that have multiple levels of memory hierarchy such as the Cell BE processor.

The second important design aspect is that VSM is a parallel programming interface that can be used by two different classes of users: compiler developers or regular programmers. VSM is a C/C++ based specification that is mainly designed to assist compilers in parallelising array expressing on target accelerators without having to deal with the underlying details of the target machine, yet it can be easily used as an Application Programming Interface (API) to operate low level communications with the machine. For example, a programmer can decompose a high-level array expression into sequences of separate operations and then call the proper VSM stub routines to parallelise one array operation at a time.

VSM was implemented in C/C++ to take advantage of the existing tools for programming the Cell processor such as GNU compilers and libraries and any future improvement of the existing tools. The VSM implementation employed two algorithms to handle alignment on virtual load and store instructions. The main ambitions in developing VSM this way are: the first objective is to provide easy access to the Cell hardware resources by hiding all the underlying details of the machine. Secondly, to use the VSM interface as an abstract model to shorten the time for developing parallelising compilers for heterogeneous systems. The third objective is to ease parallel program development by concentrating on algorithms rather than on parallelization issues such as identifying available parallelism, communication, data partitioning, alignment and synchronization.

This chapter includes a description of the VSM interface design and implementation and the challenges encountered during the development. It starts with a description of basic elements of the VSM and the messaging protocol. After that it discusses the VSM's two co-operative interpreters and the challenges encountered during the development of the VSM and concludes with the experimental results. It should be noted here that some of the material in this chapter is already published in [154].

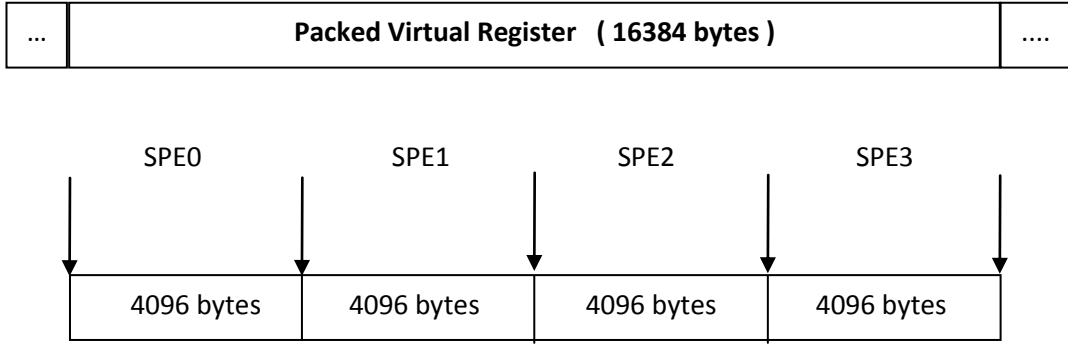


Figure 5.1: Splitting a VSM Register on 4 SPEs

5.2 Virtual SIMD Registers

5.2.1 VSM Register File

A virtual register represents a set of fields (vector data) that are logically implemented in consecutive local memory locations. The VSM design looks at data in a single virtual register as packed data which could be allocated on one SPE or scattered on multiple SPEs; see Figure 5.1. To clarify this point, the term “VSM register” shall, henceforth, be used to refer to the registers of the VSM model, and the term “SPE virtual register” shall refer to the consecutive memory locations on an SPE. Accordingly the size of the VSM registers should indicate the adequate arrays size that can be evaluated on one SPE or parallelised on multiple SPEs, and hence if a VSM register size is $PSize$ bytes, then an SPE virtual register size (S) is

$$S = PSize / P \text{ Bytes} \quad (5.1)$$

where P is the number of used SPEs.

Generally, the bigger an SPE’s virtual register size gets, the better for performing typical computation operations, such as add and multiply, on the SPEs as this allows exploiting the computing power of the Cell’s processor. It also minimises boundary checking on arrays, conditional branch instructions and the number of data transfers. Yet, due to machine alignment constrains and VSM design and performance matters, the decision on the proper virtual register size became a very important design issue, in particular for memory access instructions.

5.2.2 Size of VSM Registers

The size of an SPE virtual register is crucial in load and store operations. In the VSM implementation, these two operations depend on the Cell's DMA mechanism to move data between the Cell's main memory and SPEs' local memories. Just to recall here that the Cell processor places constraints on the size of an DMA transfer. A Cell's DMA data transfer can range from one byte up to a stream of 16-KByte [91]. Since VSM attempts to use large registers, thus the first constraint is imposed on the maximum size of data that can be transferred by a single load or store operation. The maximum data that can be moved by one SPE at a time must not exceed 16384 bytes. This implies that the maximum size of the VSM registers should be $16384 * P$ Bytes where P is the number of used SPEs. However, a virtual register size should be chosen to be large enough to pay off parallel overhead and adhere to the machine's DMA maximum size.

However, using small SPE virtual registers do not amortize communication and data movement overheads because these overheads are expected to dominate the time spent on processing small data blocks (registers). On the other hand, large SPE virtual registers require either multiple of small DMA transfers which eventually introduce additional DMA set-up times or moving a big chunk of data in a single DMA transfer, and this may result in putting an SPE's computation units in an ideal state until the data is completely transferred. Large register sizes can also limit parallelism because only arrays that have the same sizes as the virtual registers can be then parallelised.

During the development process we tested the VSM model on the Cell processor using various VSM virtual register sizes, in particular 1024, 2048, 4096, 8192, and 16384 bytes which are divided on the used SPEs. For example, if the VSM register size is 4KB, then one SPE can process 4KB, and if 2SPEs are used, then each can process 2KB and on 4 SPEs, each of which can process 1KB. Taking these considerations into account, we conducted several experiments to determine the appropriate VSM register size. Based on the current VSM implementation we found that the proper VSM register size is $4096 * P$ bytes, where P is the number of SPEs, for single-precision, 32-bit integer and 32-bit floating points, data types. More analysis on the optimal VSM register size shall be given when we discuss results and the performance of the compiler on the N-body benchmark.

The current VSM implementation also imposes divisibility and alignment restrictions on the size of SPEs virtual registers. The divisibility issue appears when multiple SPEs are used as the number of data elements in a VSM register must be divisible by the number of SPE. The other restriction has to do with data alignment to get good performance. As was mentioned before, on the Cell processor, DMA transfers must be aligned on a multiple of 16-bytes boundary, but for a better performance, they should be aligned on a 128-byte

boundary [91]. For this reason, the SPE virtual registers are all aligned on 128 bytes boundaries, yet the VSM is designed to handle moving unaligned data between the PPE and SPEs. This topic shall be discussed in detail when talking about how the SPEs handle load and store operations.

The degree of parallelism on standard SIMD machines, such as the SPEs, is determined by the number of elements of a given data type that fit in a single physical machine register, and therefore the degree of parallelism that our purposed VSM should offer is amplified by a factor of P where P again is the number of used SPEs.

5.3 Virtual SIMD Instructions

The VSM interface was designed as two layers that separate between hardware and software, and it provides access to the hardware resources and services available in a system through a Virtual SIMD Instruction (VSI) set. The VSI set is designed as high-level programming functions, and they include those aspects that are visible to users such as computation and memory access routines. The VSIs also have to collaborate with other hidden routines, such as communication and data partitioning routines, to dispatch tasks on the SPEs.

In this section, we bound our discussion only to the design and implementation of the routines that are visible to the outside world. The hidden aspects or routines, however, will be discussed in detail when we talk about the PPE and SPE interpreters.

5.3.1 Virtual Instruction Format

The VSIs are a set of RISC like register load, operate and store operations, and they are implemented as C++ functions. From now on, the word “function” will be also used to refer to a VSI. Each VSI or function consists of an opcode, one or more operands such as registers or memory locations and the effects or outcome of the instruction.

1. The opcodes determine the operations to be executed. Each instruction for each primitive data type, such as integer, float, double and characters, has a unique opcode. Table 5.1 shows some of the opcodes of primitive operations.
2. The VSIs are register-to-register instructions. An operand can be a number or a data memory location, and both must be unsigned integer values. The number would typically refer to a virtual register or a physical machine (general purpose) register

5 Virtual SIMD Machine

operation	Op-code				
	int32	int16	int8	ieee32	ieee64
load	0x01				
store	0x02				
add	0x10	0x20	0x30	0x40	0x50
subtract	0x11	0x21	0x31	0x41	0x51
divide	0x12	0x22	0x32	0x42	0x52
multiply	0x13	0x23	0x33	0x43	0x53

Table 5.1: VSM Opcodes

which are used to carry out a given operation. The memory location refers to a starting address for the load and store operations. This information has to be supplied as parameters to the corresponding function.

3. Most VSIs, which are in fact part of the PPE interpreter, have similar outcomes or tasks. These effects are:
 - a) Calculate the starting address of the data to be loaded to or stored from each SPE. The data is presumably equally distributed on the SPEs.
 - b) Combine passed parameters, the unique operation code, and the computed starting address; if needed, into message(s). This combined information determines the resources to be used in that operation.
 - c) Send formulated messages to each used SPE's inbound mailbox.
 - d) Wait for a completion acknowledgment from the SPEs (if needed).
 - e) Synchronize between different cores (if needed).
 - f) Return any expected results.

These effects, however, are completely hidden from users.

A VSI can be easily invoked by just calling the analogous function given that the required information is provided as parameters. As a matter of fact, this is the only thing a user needs to do in order to execute a VSI instruction which at the end takes all the required steps to carry out the requested operation, such as `load`, `store`, `add`, ... etc, on the Cell's SPEs.

5.3.2 Virtual Instruction Types

The VSIs set can be divided into three groups:

1. Data Movement

Data is moved between the PPE main memory and the SPEs local memories using DMA data transfers. The Load and Store instructions each requires two register operands: a virtual register and a physical PPE machine register. The virtual register is used for load or store operations while the physical machine register must contain the address which indicates the starting location of the data to be loaded from or stored in the PPE main memory.

2. Binary Computational Instructions

Most computational instructions require two operands to determine the virtual registers involved in the operation. There are, however, a few operations, such as reduction and replicate, which require a different type of registers. For example, the replicate instruction takes a scalar and makes copies of it into a virtual register, and therefore this operation also requires two register operands: one a virtual register, but the second one must correspond to a physical PPE machine register to hold the scalar value.

3. Unary Computational Instructions

These type of instructions, such as `sqrt`, `sin`, `cos` ...etc, carry out operations which usually require only one unsigned integer to indicate one virtual register number. The virtual register is used as a source and a destination register.

So far, we looked to the main design issues in the virtual SIMD instructions. We defer discussion of their implementation until we explain the hidden aspects that are connected to VSI such as the messaging protocol.

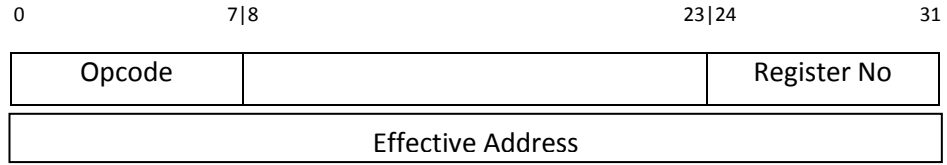
5.4 VSM Messaging Protocol

The messaging protocol facilitates the communication between the PPE and the SPEs interpreters using forward messages and return messages. These message are sent through the Cell's mailboxes.

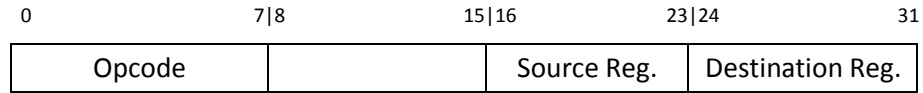
5.4.1 Forward Messages

Forward messages are mainly used by the VSIs, which are functions run on the PPE, to order the SPEs to execute an operation. These messages contain the information needed to

5 Virtual SIMD Machine



(a) Load and Store Messages Structure



(b) Computational Operations Message Structure

Figure 5.2: Message Formats

perform an operation such as an opcode, virtual register numbers, and, for Load or Store, a main memory address. The forward messages are designed in two different formats: two 32-bit word format and one 32-bit word format. The structure of the two-word format, which is shown in Figure 5.2 (a), is used in load and store operations. One of the 32-bit words as shown in the same figure contains an opcode and a register number. The first 8 bits; i.e, bits 0-7, are reserved for the opcode, and the last 8 bits; i.e., bits 24-3, are reserved for the virtual register number. The second 32-bit word holds a memory effective address which determines the starting location of the data in the PPE memory.

The one-word format, which is shown in Figure 5.2 (b), is used for other operations excluding the load and store. This 32-bit word holds three values: opcode, a source register and a destination register. The opcode must be placed in the first 8 bits; i.e, bits 0-7, the source register in bits 16-23, and the destination register number in the last 8 bits; i.e., bits 24-31.

Accordingly, the forward messages design allows encoding up to 256 operations, using 256 source registers and the same number of destination registers and addressing up to 4GB of RAM. The implementation shall be discussed in the next section.

5.4.2 Return Messages

The return messages (or acknowledgments) are sent from the SPEs to the PPE. The SPEs use their outbound mailboxes to acknowledge the PPE with operation completion. Return messages are used in operations that are required to write data back to the PPE main memory such as store or reduction operation.

5.5 PPE Interpreter

The PPE interpreter is a collection of routines that run on the master processor. These routines are implemented in C++ and can be compiled using third party C compilers for the PPE such as GNU compilers. The resulting object files along with the SPE objected file can be then linked to a user program or the output of a compiler. We shall discuss here the structure of the interpreter, main design aspects and the implementation of the main PPE routines.

5.5.1 PPE Interpreter Structure

The PPE interpreter could be divided into two subsets of library functions or routines. A set of VSM-level library routines responsible for creating and launching SPE's threads, data partitioning and communication, and a set of user-level library routines (VSIs) for scheduling data memory and computational operations to execute in parallel on the SPEs.

5.5.2 VSM-Level Runtime Library

The VSM-level library functions are completely hidden from the user, and the three main VSM-level routines: message passing, managing SPE threads, synchronisation and data partitioning. Most of these routines are implemented as inline functions to save the overheads of function calls.

- Message Passing Routine

Figure 5.3 shows the PPE function that handles two-word format messages, which are sent by user-level routines (or VSIs) to perform load or store operations. The function takes three arguments: a memory effective address (MEM_EA), a 32-bit word (msg2) and the receiver number (ID). The second argument should contain the opcode and the virtual register number. The functions also use a globally defined `p_s_area[]` array that contains the pointers of the `SPE_CONTROL_AREA` structure of all the SPEs after being mapped to the PPE address space. For more detail on the control area structure's contents see section 3.2.7.

Because each SPE's inbound mailbox can hold only four entries at any given time, the first thing this function does is to check the number of entries available in the targeted inbound mailbox by checking, within a `do/while` loop, the mailbox's status register from the `p_s_area[ID]` control area using the `SPU_MBOX_STAT` flag.

```

inline void broadCast2Msg(uint mem, uint msg2,uint ID) {
uint status,Count;
do {
    status = *((volatile uint *) (p_s_area[ID] + SPU_MBOX_STAT));
    Count= (status & 0x0000FF00)>>8;
} while ( Count < 2 );
// Assign the two messages to the MMIO registers
*((volatile uint *) (p_s_area[ID]+SPU_IN_MBOX))=mem;
*((volatile uint *) (p_s_area[ID]+SPU_IN_MBOX))=msg2;
}

```

Figure 5.3: Broadcasting PPE Messages to SPES

The do/while loop keeps iterate until the Count variable in the control area indicates that there is at least two entries available in the inbound mailbox. I had to use the do/while loop because the PPE instructions to access the mailboxes are non-blocking which means that the PPE will not stall when writing into an SPE's mailbox even if its Inbound mailbox is full, instead it overwrites the last message. Now, once the function sees the mailbox has two entries free, it then begins sending the mem and msg2 messages to the ID's inbound mailbox using again the p_s_area[ID] control area but this time with SPU_IN_MBOX flag.

The implementation of a function that sends a one-word message format is very similar to the code given in Figure 5.3; the only difference is the number of messages to be forwarded.

- Managing SPE Threads Routine

This function's role is to get the SPE ready to accept tasks from the PPE. It is called only once and must be called before attempting to use the SPEs. Figure 5.4 shows the main code of the Threads-Managing function. To shorten the displayed code, we excluded all comments, warning and error messages.

The function takes one argument, which determines the number of SPEs to be used, and has four assignments. It first reads how many SPE are available on the machine and checks if there are enough SPEs. The second task is to create one thread (contexts) per SPE and keep the pointers of the created SPEs contexts in the array T_arg[]. The third mission is to load the SPE interpreter (program) into the SPE's local store. The last assignment is to order each SPE to start running its thread.

In the last step, pthread threads were used to run the SPE contexts because the call to the function spe_context-run() is a thread blocking call. That is, if any thread launches an SPE's thread using spe_context-run() function, the launcher thread will then be blocked

```

void *Threads-Managing(uint noSPE) {
    if (spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1) < noSPE)
        exit(1);

    for(int i=0; i<noSPE; i++) {
        T_arg[i].speContext=spe_context_create(SPE_MAP_PS,NULL);
        if(T_arg[i].speContext == NULL)
            exit(1);
    }

    // load SPE interpreter into a local store
    if (spe_program_load(T_arg[i].speContext,&speInterpreter)!=0)
        exit(1);
    T_arg[i].speID=i;

    // Start running the SPE thread
    for(int i=0; i<noSPE; i++)
        if (pthread_create (&threads[i], NULL, &spe_thread_run, &T_arg[i]))
            exit (1);
}

```

Figure 5.4: SPE Thread Creation

```

void *spe_thread_run(thread_arg *T) {
    uint entry=SPE_DEFAULT_ENTRY;
    if ( spe_context_run(T->speContext, &entry, 0,0,0, 0) < 0 )
        exit(1);
    return NULL;
}

```

Figure 5.5: Launching SPE Thread Using POSIX Threads

until the launched thread finishes or is terminated. The proposed VSM model must avoid using the `spe_context-run()` function because VSM was designed to run both the PPE interpreter and the SPE interpreter at the same time. To solve this problem, VSM therefore used separate threads for calling `spe_context-run` to allow the main PPE thread and SPE threads to run simultaneously and to communicate with each other. The last loop in Figure 5.4 shows how separate `pthread` threads were used to invoke the `spe_thread_run` function shown in Figure 5.5. See section 3.2.8 for more details.

- Operations Synchronisation

Most of the VSIs or operations do not require barrier synchronisation because they were implemented as non-blocking operations in which the PPE is allowed to proceed dispatching these operations one after another. However, this is not the case with the store operation which is implemented as a blocking operation. When the

```

void OpSynch(uint noSPE) {
    uint status, Count;

    for(int i=0; i<noSPE; i++) {
        do {
            status = *((volatile uint *) (p_s_area[i] + SPU_MBOX_STAT));
            count = (status & 0x000000FF); // Extract SPU_Out_Mbox_Count
        } while ( count == 0 ); // No message posted yet

        uint ss = *((volatile uint *) (p_s_area[i] + SPU_OUT_MBOX));
    }
}

```

Figure 5.6: Barrier Synchronisation Routine

PPE orders the SPEs to store data back to the main memory, the store routine will then halt the PPE from performing any other operation until the store operation is completed.

To implement barrier synchronization, the PPE and SPEs have to collaborate with each other via messages to ensure that all the used SPEs have completed storing their data back to the main memory. Figure 5.6 shows the implementation (function) which handles this issue from the PPE side. This function expects that each SPE will send acknowledgment once it has finished the store operation. From the PPE's point of view, this function checks the outbound mailboxes of the SPEs one after another. If it finds a message in the current SPE's outbound mailbox, it pulls the posted message to empty the mailbox and continues checking the outbound mailboxes of the other SPEs. To maintain data consistency, this routine must be called by the PPE immediately after it orders the SPEs to store data back in the PPE main memory.

We also implemented this routine in a different way, trying to minimize the time spent waiting for each SPE to finish the store operation. The alternative implementation reserves a flag for each SPE and sets each flag to `false`. It then starts checking in sequence the outbound mailboxes of the SPEs whose flags are `false`. If the PPE finds a message in the current SPE's outbound mailbox, it then sets its flag to `true` to be exempted from being checked again and goes to the next. But, if no message has been posted in the current SPE's outbound mailbox at that point, it skips that SPE and checks the outbound mailbox of the next SPEs. Once it goes over all the outbound mailboxes, it then repeats the same procedure but only on the SPEs whose flags are still `false`.

5.5.3 User-Level Library Routine

The VSIs are a set of C++ functions that are available to the outside world and can be easily used to evaluate array operations on the SPEs. Using the VSIs is a one step process. Users, such as programmers or compilers, can use the VSI set by calling the analogous function providing that the required information, such as the virtual register number(s) and the starting addresses of the arrays, is supplied.

As we have mentioned in the previous section, all the VSI corresponding functions generally have similar duties such as setting messages, data partitioning, sending messages and waiting for acknowledgment if needed. For this reason, we are not attempting here to discuss every instruction individually, instead, we discuss the implementation of the VSI Store and Load operations as they includes almost all these steps that other instructions may need.

The implementation of the function that corresponds to the VSI Store operation is shown in Figure 5.7. This function takes two arguments: a virtual register and memory location, and its task is to order the SPEs to store back the contents of their virtual registers on the PPE memory. Line 1 and 2 in Figure 5.7 define the `speStoreVec()` function as an external C function. The effects of this function are:

1. Formulating Forward Messages:

This function uses a two-word message format to send the opcode, virtual register and the starting memory location. Because this information being the same for all the SPEs, this message was combined outside the `for` loop in Line 3. The statement in Line 3 uses shift operators to set the operation code and the register number in the proper bits as defined in the messaging protocol.

2. Line 4 includes a `for` loop that does the following for each SPE:

- a) Calculates the starting address

VSM model takes a simple approach in partitioning data on the SPEs. Data is equally chopped in P blocks where P is the number of SPEs. This approach may have some drawbacks due to divisibility issues. The data is partitioned into blocks of the same size in Line 5 by calculating the starting address each SPE should start from. This information represents the second message in the store operation.

- b) Sends messages

```

1: extern "C"
2: void speStoreVec(uint reg, uint mem_EA) {
3:     msg=(STORE<<24)+((reg<<24)>>24); // Formulate a Message
4:     for(int i=0; i<noSPE; i++) {
5:         mem=mem_EA+i*SIZE; // Partition Data + Set Message 1
6:         broadCast2Msg(mem, msg,i); // Passing Messages
7:     }
8:     void OpSynch() // Wait Untill Store Completed
9: }

```

Figure 5.7: Store Virtual SIMD Instruction

Once the forward messages are formulated, the PPE from within a VSI sends these messages to the SPEs inbound mailbox. Line 6 calls the `broadCast2Msg()` function to send the two messages to the i^{th} SPE.

c) Repeat steps (a) and (b) until all the SPEs are informed.

3. Process synchronisation is required in the store operation and a few other operations to ensure that each SPE has completed its part. For example, the store operation, as shown in Line 8, calls the `OpSynch()` function, to synchronise between the SPEs and also to block the PPE from issuing any other operation before the synchronisation is achieved.

Consider also the Load operation. The corresponding function is shown in Figure 5.8. It also takes two unsigned integer values: a virtual register and a memory address. Line 3 simply concatenates the value of the “LOAD” operation with the `reg` register, and in Line 4 there is a `for` loop which goes over each SPE and does the following: it calculates in Line 5 the starting address from which each SPE should start loading data into the `reg` register, and then it calls in Line 6 the `broadCast2Msg` function to send the two messages to each SPE. Notice here that once the PPE delivers the messages, it can then proceed in executing other operations, and that is why the Load and most computational operations are called non-blocking operations.

5.6 SPE Interpreter

The SPE interpreter is a C++ program that runs in parallel on several SPE’s to execute micro-tasks. The program runs constantly in a loop on each SPE in the background and frequently checks if there is any message dispatched by the PPE.

```

1. extern "C"
2. void LoadVec(uint reg,uint mem ) {
3.     msgs[0]=(LOAD<<24)+((reg<<24)>>24);
4.     for(int i=0; i<noSPE; i++) {
5.         mem=mem_EA+i*SIZE; // Partition Data + Set Message 1
6.         broadCast2Msg(mem, msg,i); // Passing Messages
7.     }
8. }

```

Figure 5.8: Load Virtual SIMD Instruction

5.6.1 SPE Interpreter Structure

The Algorithm 5.9 outlines the main steps of the SPE program. It is basically constructed of a large switch statement which contains a case for each opcode (operation). The first two jobs in the algorithm are: get any messages in the SPE's inbound mailbox and then decompose the received messages to identify the requested operation (opcode) and its operands. The main job of the SPE interpreter is then handled by the `switch` statement which accordingly chooses the analogous operation. The computational operations are straightforward routines, but the memory access operations, load and store, are complicated and challenging due to data alignment and synchronisation capacities.

In the following discussion, we first look at the implementation of the first two steps. We then talk about the main design challenges, techniques and algorithms that we have used in load and store operations. After that, we present the implementation in the Load and Store operations and two samples of computational operations.

5.6.2 Extracting Information

The first two steps in the SPE program are very important because they affect both the PPE and the SPEs executions. The implementation of these two steps is shown in Figure 5.10. In the first statement, the `spu_readch()` function with the `SPU_RdInMbox` flag reads an inbound mailbox channel. This operation is a blocking operation which means that the SPE stalls when the queue is empty until it receives a message from the PPE. Upon receiving a message, the interpreter extracts the opcode and the register, and then check if

```

while (1) {

    1. Pull the messages // blocking mode

    2. Extract information //opcode, registers and MEM_EA

    3. switch (opcode) {
        a) case LOAD:
            i. Alignment
            ii. Load Virtual Register
        b) case STORE:
            i. Alignment
            ii. Store Virtual Register
            iii. Data Synchronisation
            iv. Acknowledgement
        c) case ADD:
            i. Execute Operation
        d) .....
        e) .....
        f) case TERMINATE:
    }

}

```

Figure 5.9: SPE Interpreter Structure

```

1. msg=spu_readch(SPU_RdInMbox);
2. opcode=(msg>>24);
3. Reg1=((msg<<24)>>24);
4. if ( opcode < 8 || opcode==69)
5.     mem_ea=spu_readch(SPU_RdInMbox);
6. else
7.     Reg2=((msg<<16)>>24);

```

Figure 5.10: Messaging Pulling Code Segment

the extracted opcode is store or load, it should then read the second message which holds an effective address on the main memory.

5.6.3 Load Operation

The VSM Load operation brings data from the PPE main memory into the SPE's local memories using DMAs. It is a non-blocking operation which assumes that the compiler will not get the PPE to write into the area being loaded. The non-blocking mode allows the PPE to continue its work on the current expression after it delivers the two messages of the load operation to the SPEs. The PPE also does not need to wait until the data is loaded into the SPE's local memories because the SPE can check the completion status of any DMA involved in loading a virtual register. The only problem with the Load operation is the DMA alignment constraints on the starting memory locations. The addresses on both PPE and SPEs' sides must be aligned to the same boundary. Thus, having all the SPEs local buffers (vector registers), as in our VSM, aligned to 128 bytes is not enough because there is no guarantee that data transferred from the main memory is aligned. There is, in fact, a high probability that the data to be transferred is unaligned, specially when dealing with sub-ranges of an array or 2D arrays. For this reason, we developed an algorithm for loading aligned or misaligned data to the SPE aligned registers.

To illustrate the algorithm let us define:

- MEM_E and MEM_A as Effective and Aligned addresses on the main memory respectively.
- LS_E and LS_A to be Effective and Aligned addresses on an SPE's local storage respectively.

Algorithm 5.1 Alignment Load

1. Get effective address(MEM_E)
 2. $AB = \text{MOD}(\text{MEM_E}, \text{CLS})$
 3. $\text{MEM_A} = \text{MEM_E} - AB$
 4. If $AB = 0$
 5. Transfer VR_S bytes start from MEM_A into LS_E
 6. else
 7. Transfer VR_S+CLS bytes start from MEM_A into TB
 8. Copy VR_S bytes start from TB+AB to LS_E
-

- VR_S is the size of a virtual register.
- CLS is the target machine's Cache Line Size in bytes. On the Cell, CLS=128;
- TB is a temporary buffer on the SPE, and its size is VR_S+CLS,
- T is a temporary buffer of size CLS bytes.
- Also assume that each register name represents its starting address on an SPE's LS and that all DMA transfers are aligned on the CLS boundary on both sides.

Given the above definitions, which will be also used in the discussion of the store instruction, we present the algorithm shown in Figure 5.1 to load aligned or unaligned data of any data type. The algorithm works as follows. If data to be transferred is aligned, then the algorithm will skip to Line 5 which loads VR_S bytes of data starting from MEM_E into the SPE local memory (LS_E). However, if data (MEM_E) is not aligned, then Line 2 calculates the offset, and in Line 3 it sets MEM_A to point to the first aligned location before the effective address. In Line 7, it grabs VR_S+CLS bytes of data starting from the calculated aligned-location (MEM_A) and copies these bytes in the TB temporary buffer. Transferring additional CLS bytes ensures that the first CLS bytes will include the MEM_E and that the MEM_E+VR_S will be within the last CLS bytes in the TB buffer. Now, to get rid of these additional bytes, Line 8 copies only VR_S bytes from the TB temporarily buffer but it would start from the TB+AB address.

Figure 5.11 shows an example of loading 1024 unaligned bytes from the PPE main memory into an SPE virtual register (local memory). Just to recall that the Cell's cache line size CLS is 128 bytes.

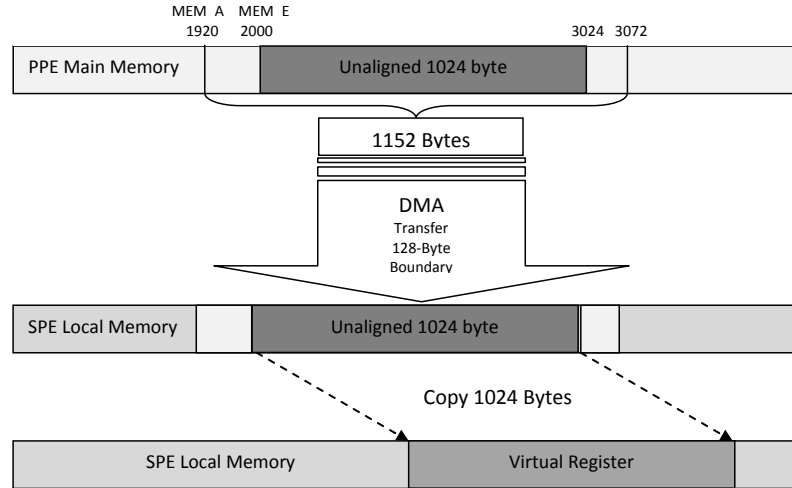


Figure 5.11: Loading Unaligned Data to SPE's Local Memory

5.6.4 Store Operation

The Store operation is even more complicated than the Load operation for a couple of reasons. First, this operation transfers data back from the SPEs to the PPE main memory using DMAs, and therefore, in keeping with the alignment constraints on the DMA transfers; some parts of the data are expected to be shared among the SPEs or between one of the SPEs and the PPE. The second reason is that the store operation is a blocking operation, and thus the SPEs need to collaborate with the PPE for barrier synchronisation.

In the Cell, each SPE includes a globally coherent DMA engine [136, 91]. DMA transfers offer coherent data operations to ensure only ideal data sharing between the Cell's cores [155]. When transferring data from the main memory to an SPE local memory, the MFC will sneak the data from the PPE's cache if it contains the most recent data. Similarly, when transferring data from an SPE local memory to the main memory, the cache line which is associated with the transferred data blocks are invalidated and therefore any future access to these locations gets updated data.

However, our VSM depends upon DMA transfers that do not act a globally coherent cache in the standard sense, and therefore it must perform coherence maintenance to ensure data consistency when it attempts to write back unaligned data in the main memory. Writing back unaligned data using DMA transfers from the SPEs requires reading and merging bytes from the main memory before rewriting those bytes as part of a DMA. This problem arises when multiple SPEs are used or even if one SPE was used. Based on the VSM design, every virtual register is chopped into blocks to be distributed on the available SPEs, and these blocks are not necessarily aligned. This implies that the SPEs will work on adjacent unaligned blocks of data into main memory or locations near to each other,

and therefore attempting to update these locations concurrently using, for example, aligned 128byte DMA transfers may result in collisions between the SPEs. This situation could also happen between one SPE and the PPE. For example, consider an SPE that attempts to cache a block of data using a single DMA transfer. If the data is not aligned, the SPE in this case has to read some bytes from the main memory to adhere to DMA alignment constraints. Yet, this requires that the additional bytes, which have been just read, remain unchanged when written back into main memory as a part of the issued DMA transfer(s), potentially overwriting only the changes that are data processed by the SPE.

The alignment problem in the store operations is critical as it may result in race conditions and data inconsistency. To solve these issues, I developed an algorithm for storing unaligned data from one or multiple cores. The main concept of our algorithm is to use lock-based synchronisation. This type of synchronisation first ensures consistency of data that might be shared due to alignment constraints. Secondly, it avoids any race conditions that might occur, and lastly it minimises the portion of data that is unavailable to other processes.

5.6.4.1 Store Processing Algorithm

The algorithm, as shown in Figure 5.2, is based on an operation-dividing technique in which each SPE's data block is divided into three parts (or data sub-blocks): Head, Middle and Tail. According to our proposed parallelisation technique, only the two ends, Head & Tail, would be shared between the Cell's cores, and therefore separating the middle part allows a DMA transfer to be used for each part, to work easily on the unshared part and to enable concurrent data transfers.

The key design elements in the algorithm are:

- Use a separate DMA transfer for each part: Head, Middle and Tail .
- The Head and Tail length depend on the machine resources and limitations, but they should be kept as small as possible.
- Data will be shared between a maximum of two cores: the PPE and one of the SPEs or between two SPEs.
- Let the size of each end be equal to CLS .
- Only one SPE at a time can obtain lock on all CLS bytes of memory or even on part of them.

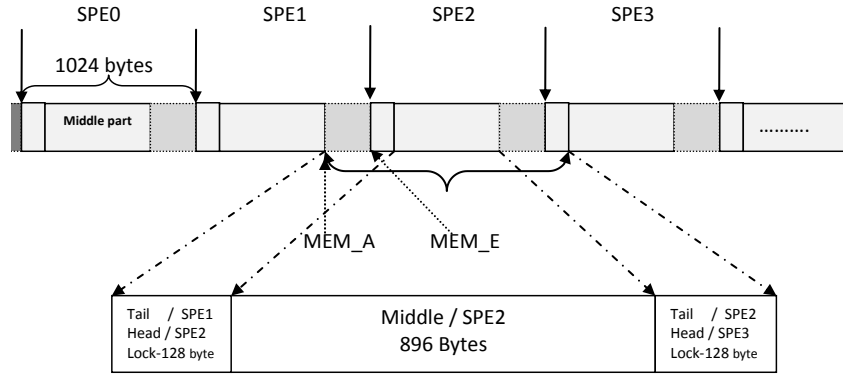


Figure 5.12: Splitting an SPE Block into 3 DMAs for Storing Process

- If multiple SPEs want to obtain lock on any part of the data, one SPE wins and the other will wait.
- For both the Head and Tail, set a lock on CLS bytes, which should cover one end, read, overwriting changes made by the SPE and then unlock.
- Write back the middle part into the main memory.

The algorithm shown in Figure 5.2 first computes the aligned memory location (MEM_A) that is preceding the MEM_E effective address. It then starts working on the three different parts: Head, Middle and Tail.

Figure 5.12 illustrates how the algorithm is applied on the Cell processor. The size of the Head and Tail was chosen to be 128 bytes for two reasons: First, all DMA transfers are aligned on a 128-byte boundary, and secondly the Cell offers atomic DMA operations, such as `getllar()` and `putllc()`, to set, reserve and release locks on 128 bytes; for more details see section 3.2.2. Now since the Head and Tail are 128-byte aligned blocks, the Middle portion size should be 128 byte smaller than the virtual register size; that is, $VR_S - 128$ bytes.

5.6.4.2 Implementation

The following two figures show the main parts in the implementation of the store algorithm for the Cell processor. The code was split into two parts just to be readable and understandable. Figure 5.13 shows the initialization part and the code for storing the Head of an SPE data block. The Tail would also be coded similarly a side from the starting addresses. Note that in this implementation the Head and the Tail each have 128-byte length; i.e., $CLS=128$.

Algorithm 5.2 Unaligned Store Operation

```

1. AB = MOD(MEM_E, CLS)

2. MEM_A = MEM_E - AB

   // HEAD (PARTIALLY UPDATED)

3. SET LOCK ON CLS BYTES START FROM MEM_A

4. GET CLS BYTES START FROM MEM_A INTO T

5. APPEND THE RESULTS TO THE TAIL OF T

6. PUT BACK T, AND RELEASE THE LOCK

   // UPDATE THE MIDDLE PART (COMPLETELY UPDATED)

7. MEM_A = MEM_A + CLS

8. LS_E = LS_E + CLS-1 - AB

9. MS = VR_S - CLS

10. COPY MS BYTES START FROM LS_E TO LS_A

11. PUT MS BYTES BACK START FROM LS_A TO MEM_A

   // TAIL (PARTIALLY UPDATED)

12. MEM_A = MEM_A + MS

13. SET LOCK ON CLS BYTES START FROM MEM_A

14. GET CLS BYTES START FROM MEM_A INTO T

15. APPEND THE RESULTS TO THE HEAD OF T.

16. PUT BACK T, AND RELEASE THE LOCK

```

```

// Initialization
uint AB, MEM_A, LS_A;
AB = MEM_E%128;
MEM_A = MEM_E-AB

// Store Head
do{
    mfc_getllar((void *)T, MEM_A, 0, 0); // Lock & Read 128byte
    (void)spu_readch(MFC_RdAtomicStat); // Reserve the Lock
    memmove((void *)((uint)T+MEM_A), (void *)LS_A, CLS-MEM_A);
    mfc_putllc((void *)T, MEM_A, 0, 0); // Store and release lock
} while(spu_readch(MFC_RdAtomicStat)&MFC_PUTLLC_STATUS);

```

Figure 5.13: Synchronise Shared-Block

Figure 5.14 shows the code to store a Middle part of an SPE data block. This code should be executed after storing the Head portion. The first line shifts the MEM_A pointer to point to the first aligned location in the PPE main memory that comes after the Head; that is, the starting address from which the Middle part should be stored. The second line calculates the starting location of the data block which resides on the SPE local memory. This location must come 128-AB bytes from the beginning (LS_A) of the block. The third line checks if the new location of local data is aligned. If it is not, the Middle part then must be copied in an aligned buffer, called here TMP_REG, on the SPE local memory, before moving it to the memory. This step necessary as the source and destination addresses in any DMA transfer must be aligned to at least a 16-byte boundary. The last step is a standard DMA function, `mfc_put()`, that transfers data from SPE local memory to the PPE main memory. After the two addresses in the function are ensured to be aligned, the standard `mfc_put()` function can now be safely executed without the need for any locks or synchronisation.

So far, we have discussed how the store operation handled data synchronisation, but it also has to collaborate with the PPE implementation to maintain barrier synchronisation as we have just mentioned above. In the store operation, the SPEs are required to notify the PPE with the completion once each SPE ensures that its entire data block has been stored into the main memory. Figure 5.15 shows the two lines of code that handle the acknowledgment. These lines must be executed by the SPEs at the end of the operation. The `spu_readchcnt()` function with `SPU_WrOutMbox` flag in the first line checks when the outbound mailbox is empty. and the second line will write a message to the outbound

```
// transfer Middle part of VSM register
MEM_A=MEM_A+128;
LS_A=LS_A+128-AB;
if (LS_A%16 != 0 ) { // Unaligned local Address
    memmove((void*)((uint)TMP_REG),(void*)LS_A,SIZE-128);
    LS_A=TMP_REG;
}
mfc_put((void *)LS_A, MEM_A, SIZE-128, destReg, 0, 0);
```

Figure 5.14: Storing Middle Block

```
do{}while(!spu_readchcnt(SPU_WrOutMbox));
spu_writech(SPU_WrOutMbox, 99);
```

Figure 5.15: Acknowledgment of an Operation Completion

mailbox once it is empty. The message could be any 32-bit word, but we used the number 99. The PPE can validate the completion by checking if it received the number 99 via the mailbox or not.

5.6.4.3 Design Challenges

In the alignment algorithm shown in Figure 5.12, the Head and Tail of an SPE's block are apparently shared between the Cell's cores. And to ensure that these shared parts of data are over-written in a proper order, we used locks to update the Heads and Tails. This synchronization process keeps memory coherent and avoids any race conditions that could arise as different SPEs attempt to update the Heads and the Tails. The race condition problem is solved by reserving a lock on each 128byte until the granted SPE updates the data and then releases the lock.

The algorithm for storing unaligned data was designed with the intention of optimising the cost due to the DMA overheads and the extra memory copies we had to do using the `memmove()` function. The order of these DMAs, as presented by the algorithm in Figure 5.2, provides some overlapping within one SPE and among the SPEs. The overlapping within one SPE, for example, occurs while storing the Middle part and the Tail. Once an SPE issues a DMA request to its MFC for transferring the Middle part, the MFC takes control of the transferring process; meanwhile the SPE continues with processing the Tail by requesting a lock...etc. Once the Tail is stored back, the SPE should then verify if the Middle part has been completely transferred. The DMA's order also allows SPEs to overlap updating the Heads and Tails. For example, when SPE0 is locking its Head, SPE1 can be easily granted a lock on its Head too because the SPE1's Head is part of the

SPE0's Tail, and therefore by the time SPE0 reaches the points to update its Tail, the SPE1 would most likely be finished from updating its Head.

5.6.5 Computational Operation

The VSM is basically designed to use the SPEs to perform computational operations on one dimensional arrays (vector registers) or scalars such as in the replicate and reduction operations. Most of these operations, however, have similar implementations. This subsection first describes the implementation aspects that are common in these operations and then looks at three operations that have some differences in the implementation.

The common aspects in the implementation of these operations are:

- All the operations are carried out on the SPEs using built-in SPU intrinsic functions. These intrinsic functions, which support all primitive data type operations, operate on a 128-bit at the time using SIMD instruction set to obtain better performance.
- An SPE's virtual registers (data buffers) are basically defined as vectors of character data type.
- The character vector registers are converted to the proper data type once the operation is known.
- A unary operation requires only one vector register which is used as a source and a destination.
- A binary operation requires two vector registers: a source and a destination.
- The PPE supplies the SPEs with the vector register numbers for arithmetic operations such as Add and Subtract.
- The PPE supplies the SPEs with a vector register number and a reference to a scalar for operations such as replicate.
- Virtual registers are cast to vectors of corresponding primitive data types, such as (vector float *), in order to use the intrinsic functions.
- Virtual registers must be length equal to 128 byte or a multiple of 128 byte.
- The number of iteration ITER is an unsigned integer that equals

`REG_SIZE*Sizeof(dataType)/128.`

```

case ADDF: {
    vector dataType *aptr=(vector dataType *) VSM_REG[destReg];
    vector dataType *bptr=(vector dataType *) VSM_REG[srcReg];
    for (int j=0; j < ITER ; j++)
        aptr[j]=spu_add( aptr[j] , bptr[j]);
    break;
}

```

Figure 5.16: Add Operation

The ITER variable determines the number of SIMD operations required.

- The VSM_REG[] is an array of pointers that point to aligned-data buffers (vector registers) on the SPEs.

5.6.5.1 Add

The Add operation is a basic binary non-blocking operation. It takes two vector registers, adds these vectors element by element and stores the result in the first register. Figure 5.16 shows a genetic implementation of the Add operation.

5.6.5.2 Replicate

The Replicate operation is also a binary operation, but it takes one vector register and a scalar instead of two vector registers like in Add, subtract, multiple, ...etc. This operation basically takes a scalar value and copies it in each element of the vector register. Figure 5.17 shows the implementation of this operation. The major difference from standard binary operations is due to the scalar operand as it needs to be handled differently from vector registers. When an operation requires a scalar, the PPE sends only the main memory location of that scalar to the SPEs which must subsequently consider any alignment measures.

In Figure 5.17, the code for aligning a scalar value was placed in a separate function, called `alignScalar()`, just to be more clear and understandable. The `alignScalar()` function takes the following two main steps. First, it copies the aligned 128 bytes including the location (MEM_E) of the scalar into the SPE local memory. It then adjusts the scalar pointer to point to the exact location of the scalar value which was copied on the local memory.

```

case REPF: {
    vector dataType *aptr=(vector dataType *)VSM_REG[destReg];
    dataType *bptr=(dataType *) alignScalar(MEM_E);
    for (int j=0; j < ITER ; j++) {
        aptr[j]=spu_splats(bptr);
        break;
    }
}

void *alignScalar(uint MEM_E) {
    char TMP[128] __attribute__((aligned(128)));
    unsigned int AB=MEM_E%128;
    void *scalar __attribute__((aligned(128)));
    mfc_get(TMP, MEM_E-AB, 128, 1, 0, 0);
    scalar=(void *)((uint)TMP + AB);
    return scalar;
}

```

Figure 5.17: Replicate Operation

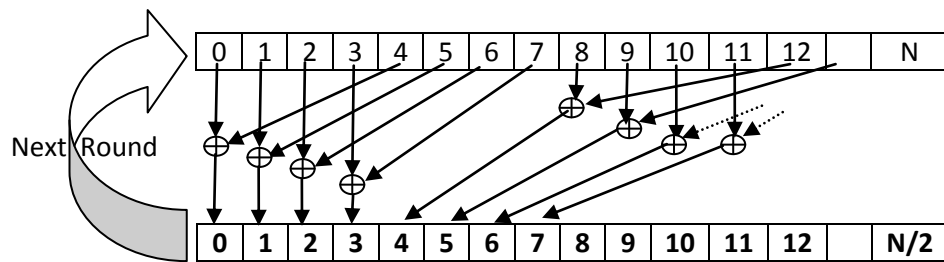


Figure 5.18: Pattern Of Reduction Operation

5.6.5.3 Vector Reduction Operation

The VSM also supports associative reduction operation on the SPEs. Reduction operations defer from other computational operations in several aspects. They are blocking operations unlike other computational operations, and they are also unlike Replicate operations from the input/output prospective. A Reduction operation's outcome is a scalar value while the Replicate operation's outcome is a vector register.

The VSM reduction operation is based on a recursive vector reduction method. Using standard scalar instruction and registers, the method basically starts with a vector of length equal to $N=VEC_SIZE$ and basically adds two consecutive elements together one after another and keeps the results in the same vector. The number of addition operations is $N/2$ and the number of elements is reduced to half of each iteration ($N/2$). In the second iteration, the intermediate ($N/2$) results are again reduced to half; $N/4$ elements. The final scalar result will be obtained after $\log_2 N$ iterations or rounds.

However, our VSM depends on SPE intrinsic functions to carry out arithmetic operations, and therefore the number of consecutive elements that are added to each other at a time is $\text{REG_SIZE}/\text{Sizeof}(\text{dataType})$ where REG_SIZE is the size of the physical machine's registers in bytes. Figure 5.18 illustrates how elements of a 32-bit floating point vector are added together using 128-bit SIMD instructions; i.e, $\text{REG_SIZE}=128/8=16$ bytes. Give that

$$N = \text{VEC_SIZE} * \text{Sizeof}(\text{dataType})/16$$

the number of operations needed to evaluate a vector of size VEC_SIZE elements of type "dataType" using 128-bit SIMD instructions at the first round (iteration) is $N/2$, and thus the final scalar result will be obtained after $\text{Log}_2 N$ iterations or rounds.

However, if the SPE intrinsic functions are used, then there will be an intermediate result which resides in a vector of size

$$\text{REG_LEN} = \text{REG_SIZE} / \text{Sizeof}(\text{dataType})$$

This intermediate result must be then added together element by element as shown in the second loop structure in Figure 5.19 and then sent to the PPE. Figure 5.19 shows the implementation, which uses the same variables given in the above discussion, of the Reduction operation using the `spu_add()` intrinsic functions. The code in this figure shows also how the result is sent back to the PPE using the `mf c_put()` function and the SPE acknowledgment to the PPE via the outbound mailbox.

5.7 Using VSM

The VSM tool imitates SIMD concepts to increase the work a single instruction performs. The VSM implementation hides all the underlying hardware and software of the Cell processor, automatically parallelising array operations and supporting scalable parallelisation. It basically consists of two co-operative interpreters that are implemented in C++ and each contains a number of functions and routines. The PPE functions have to establish the communication with an SPE, choose the proper operation to be performed on the SPEs, dispatch the request to the PPE via messages, and handle any synchronisation required. On the SPE side, there is an SPE program, which runs in the background and checks if there is any message deposited into its Inbound mailbox. Once an SPE receives a message, it then starts dealing with the requested operation taking into consideration any alignment and data synchronisation issues.


```

case REDFP: {
    // REDUCTION OPERATION (dataType)
    vector dataType *aptr=(vector dataType *)VSM_REG[destReg];
    uint j=0;
    while (N > 1) {
        for (uint i=0; i < N ; i=i+2)
            aptr[j++]=spu_add( aptr[i] , aptr[i+1]);
        N=N/2; j=0;
    }
    dataType *a=(dataType *) VSM_REG[destReg]; sum=0.0;
    for (uint i=0; i < REG_LEN ; i++)
        sum+=a[i];
    mfc_put((void *)&sum, (void *) MEM_E, 128,1,0,0);
    do{} while(!spu_readchcnt(SPU_WrOutMbox));
    spu_writetech(SPU_WrOutMbox,(unsigned int)REDFP);
    break;
}

```

Figure 5.19: Vector Add Reduction Operation

```

spu-g++ SPEcode.cpp -lsimdmath -o SP
ppu32-embedspu speInterpreter SP spe.o
ppu32-g++ -c PPEcode.cpp -o spe.o ppe.o

```

Figure 5.20: Building VSM Object Files

VSM can be built using only three command lines in which third party C++ compilers for the PPE and SPE processors are used to compile and link both interpreters. Figure 5.20 shows these command lines. The first line compiles the SPE interpreter, called `SPEcode.cpp`, in the `SP` binary. The second command embeds the `SP` binary file into the PPE compatible object format `spe.o`, and `speInterpreter` is the name used in the PPE to call the SPE's binary. The last command compiles the master interpreter, `PPEcode.cpp`, and builds the object file `ppe.o`. Now, the PPE and SPE object files are ready to be linked with any program to evaluate array operations on the SPEs.

VSM was essentially designed to be used as an interface to extend array programming language compilers for automatic parallelisation, but it can be used as a fully implicit programming model for parallelising array operations on the SPEs.

5.7.1 Code Generator Interface

Developing the VSM as a programming tool that can assist compilers in parallelising code on heterogeneous architecture was the main objective of this work. Compiler developers

can use VSM as an intermediate layer or an interface to access the Cell's SPEs. A compiler that attempts to use VSM is expected to have the capability and flexibility to generate code with a degree of parallelism that is already supported by the VSM implementation. If a compiler has such capability, it then can automatically parallelise array expressions by generating the code which can invoke the VSM instruction set. This usually requires defining a new instruction set that the compiler code generator can use. The following chapter shall talk about how the VSM implementation was employed to extend the Glasgow Vector Pascal (VP) compiler to the Cell BE processor.

5.7.2 Application Programming Interface (API)

The VSM implementation can also be used as an API. A programmer can decompose a high-level array expression into sequences of individual operations and then call the proper VSM stub routines to perform in parallel one array operation at a time on the SPEs. Though, the VSM current implementation has not been tuned yet to use as an API, it can be very easily used by programmers. Figure 5.21 shows a simple C++ program `foo.cpp` that uses the VSM routines to perform some operations on 2 SPEs. An identical copy of this program was compiled, linked and ready to be executed by using only the following command.

```
ppu32-g++ foo.cpp ppe.o spe.o -lspe2 -lm -o
```

Note that the program does not include any annotations or directives, and the programmer does not need to handle SPEs thread creation, data partitioning, data transfers, alignment and synchronisation.

5.8 Experimental Results

We present here the results obtained from testing the VSM implementation as an independent programming tool. Chapter 8 includes further results which show the performance of the VSM on BLAS benchmarks and real-world application after being integrated with a VP compiler. All the tests, which are presented here, were carried out on a Sony PlayStation 3 console, running Fedora Core 7 Linux and using the IBM Software Development Kit (SDK) v3.0.0. The PS3 has a single Cell chip. The chip includes a 3.2 GHz master processor which has 256MB main memory and 6 SPEs with 256KB local memory and the same speed. The VSM implementation is compiled, assembled and linked using `ppu-g++` and `spu-g++` (GNU tool chain v4.1.1).

```

#include <stdlib.h>
#include <stdio.h>
#include "PPE.h" // Includes Functions Prototypes
const int N=4096;
float v1[N],v2[N],S;
int main() {
    speInitialize(); // Creat,load & launch SPE Threads
    for (int i=0;i <N;i++)
        v1[i]=1.1; // Initialize Vector v1
    RepVec(0,1.25); // Replicate 1.25 in SPE register 0
    LoadVec(1,(uint)v1); // Load v1 in SPE Register 1
    MulVec(0,1); // Cross Product SPE registers 0 & 1
    StoreVec(0,(uint)v2); // Copy the result into vector v2
    S = redpf(v2,N); // Vector Reduction Add operation
    printf("\n\n The result of Reduction=%f \n\n",S);
    speEnd();
    return 0;
}

```

Figure 5.21: A C Program Uses VSM as API

The following sections discuss the VSM key operations that we attempted to assess using two single precision floating-point vectors. The discussion starts with a simple code just to show how to invoke the VSM four key functions (operations): Thread creation, Load, Store and a computational operation. In this discussion, we identify the main activities or parameters in these operations. After that, it discusses the performance of the different operations and the activities involved. Note that all the measures were taken from the PPE.

5.8.1 Simulator

In order to use the VSM implementation before it was integrated with the VP compiler extension, we had developed a simulator to imitate a generated-compiler code. The simulator is a simple C program that uses the VSM as an API. The code in Figure 5.22, for example, shows the main part of a simulator that adds to two vectors *v1* and *v2* and stores the result in vector *v3*. We also appended a few lines of code into the PPE functions to time the different activities of each VSM instruction, such as communication, computation and alignment. The appended code is not shown here since it involves only code for calculating execution times.

5.8.2 SPE Thread Creation

In order to use the SPEs, this operation must be called first. It involves three activities:

```

//Create and Launch 2 SPE threads

    speInitialize(2);

// Load vector v1&v2 from PPE into SPE's virtual reg. VR0&VR1

    loadVec(0,v1);
    loadVec(1,v2);

// Add the SPEs virtual reg. VR0&VR1 and keep the result VR0

    addVec(0,1);

// Store the contents of VR0 into vector V3 (main memory)

    storeVec(0,v3);

// Terminate the SPE threads

    speEnd();

```

Figure 5.22: C Simulator

- Create an SPE thread.
- Load the SPE program.
- Launch the thread.

To measure the latency time of creating one thread we run this operation to first create one thread on one SPE, 2 threads on 2 SPEs and then 4 threads on 4 SPEs. Table 5.2 presents The average latency of the basic thread management activities involved in creating threads on the Cell accelerators.

Number of Threads	Average latency of managing SPE threads (msec)			
	Creating	Loading SPE	Launching	Average
1	0.70	0.17	0.12	1.99
2	0.65	1.05	0.09	1.78
4	0.65	0.99	0.06	1.69

Table 5.2: The average latency of the basic thread management operations on the Cell accelerators

Figure 5.23 (a) shows that the average latency for setting up a single thread ranging from 1.68 msec to 1.98 msec depending on the number of SPEs. The timing includes the three steps or activities, create, load and launch, as well as the time required to ensure that each activity has completed. Similar latency had been also obtained by Abellan et al. [74] in which the operation was run 10^5 times.

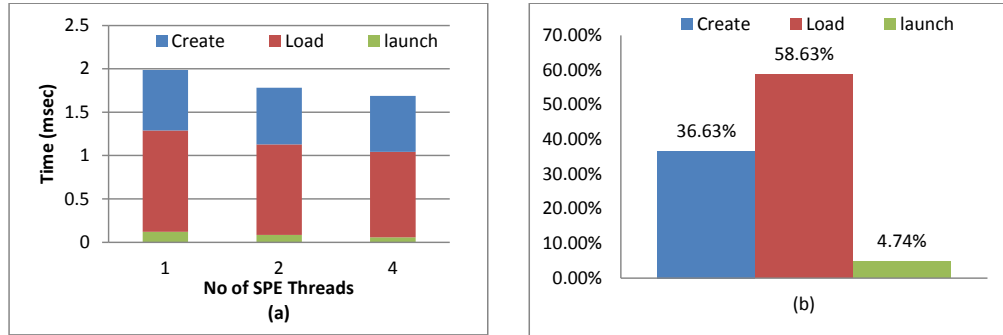


Figure 5.23: (a) shows the latency per thread Launching while (b) approximates the percentages of time spent in the activities involved in setting up a single SPE thread

The graph (a) plots the average latency time for setting up one thread when 1, 2 and 4 SPEs were used. As we can see, the average time needed to set up one thread decreases as the number of SPEs increases. Actually, the average latency for creating or launching a single thread is almost the same even when the number of the threads increases, but the latency for loading the code on the SPEs decreases as their number increases. The interpretation for this is that there is some overlap between loading the SPEs' programs which saves some time.

Figure 5.23 (b) charts the percentage of the three activities as compared to the average of the total latency. This diagram clearly shows that loading an SPE code dominates the thread creation process. It takes approximately 36.5% of the total time to create an SPE thread while around 58.5% of the time spent loading an SPE program. It also is very interesting to notice that it takes about 4.75% of the time only to launch an SPE thread. This was explained in steps 4 and 5 in Figure ???. Actually, the PPE is only required to request, step 4, an SPE's MFC to begin executing the loaded thread, and the MFC handles the request internally as stated in step 5 in the same figure.

However, in order to reduce the thread creation latency, the compiler code generator can append a call to the function, which creates the SPE threads, at the prolog (start up) session. It can also order to terminate the threads when the application finishes by appending a call to the terminating function at the Epilog session. This keeps the threads alive until an application is completed.

5.8.3 Messaging Using Mailboxes

The VSM depends mainly on the Cell's mailbox mechanism using MMIO registers to exchange information between the different cores. Every VSM instruction (PPE function)

is required to communicate with the SPEs through their mailboxes in order to hand in its request to the SPEs. We used this mechanism because all the available documents state clearly that the mailbox mechanism is much faster than using DMA transfers to exchange small data sizes [136, 54, 57, 135, 91, 56]. However, at the first stages of the VSM development, it was very hard to prove that the mailbox's mechanism is faster than DMA and Signal mechanisms.

At the beginning, we started investigating the latency of different operations and in particular the latency of sending a message between the PPE and an SPE. In our experiments, we used the *spe_*_mbox** function from the SDK library to access mailboxes easily [135]. This SDK function allows the PPE to read or write to a mailbox or check the status of a mailbox. Yet we encountered a critical problem as all the experimental studies showed that the mailbox mechanism is very slow. It is sometimes even slower than DMA transfers if we compare, for example, the average time to send 32 bytes using mailbox or DMA transfer, especially if the later was used to move a large data block. However, the resources, which were available during the first stages of our VSM development, reported that the PPE can use the SDK library function or write immediately into the SPE's MMIO registers, but they did not mention that the SDK function is slower than using the later approach.

In April 2009, I attended a 2 day training course at Daresbury Science and Innovation Centre in Warrington, UK, and I had the chance to raise the problem with two of the manufacturer engineers who recommended to try using MMIO registers. I then asked them if they think the MMIO is much faster than the SDK functions. I also mentioned to them that all the measures that I had at that time showed that a mailbox message using the SDK functions costs around 0.01 millisecond. They said it is expected, but they do not know how much faster the MMIO register approach could be as compared to the SDK function. After the training course finished, I re-run all the tests using MMIO registers instead, and surprisingly the test shows that the MMIO mechanism is at least 12 times faster than the SDK function. About the same speed up was also reported by a paper published around the same period which characterises the basic communication operations using dual Cell-based Blades [156]. The main reason behind the increased latency is that the SDK function is a runtime management function that involves a system call which is quite costly.

It is important to explain here, before reflecting on the timing, the PPE procedure which was implemented to send a message to one or more SPEs. To send a message through MMIO registers, the PPE must first check the number of entries available in the target mailbox because it will not stall even if the mailbox is full and overwrites the last message. If more than one SPE is used, then there are two procedures to do that. In the first

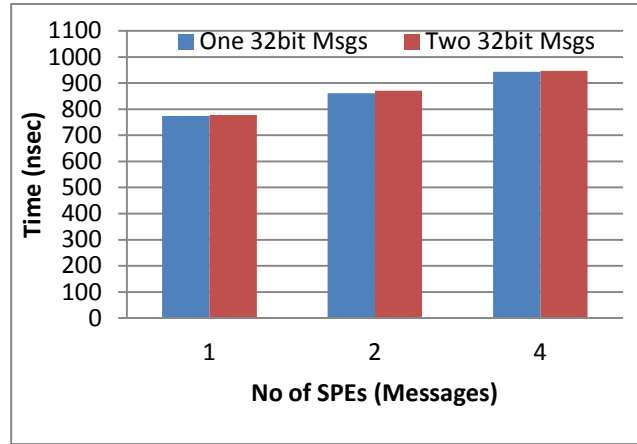


Figure 5.24: The Latency of Sending Messages to SPEs

procedure, the PPE checks the mailboxes one after another, and it does not go from one SPE to the next unless it sends the message to the current SPE. The second procedure is meant to reduce the time to wait for each SPE's mailbox to be empty. In this procedure, the PPE uses polling techniques to check if the first SPE's mailbox has enough space to hold the messages. If the checked SPE has enough space, the PPE then posts to it the message(s); otherwise the PPE goes to the next SPE's mailbox and so on. That is, the PPE does not have to wait to get free entries in a given mailbox instead it skips the currently checked mailbox to another SPE whose mailbox is free until all the SPEs receive the messages. This procedure is used by all VSM operations, such as load, store and computational operations, to send the PPE request to the SPEs, and therefore the cost of sending a message is the same in all operations.

This operation was run 10^5 times to send one 32-bit message and two 32-bit messages to 1, 2 and 4 SPEs. Figure 5.24 shows the average latency time of sending messages in nanoseconds. The time to send a one or two word message from the PPE to one SPE is almost the same; it is around 780 nsec. To send the same message to two SPEs, the latency increases by about 90 nsec as compared to one message and by another 85 nsec with 4 messages to four 4 SPEs. The additional process to check each SPE mailbox status and the loop iterations needed to go over the SPEs was behind these increases.

5.8.4 VSM Instructions Performance

The following discussion focuses on three key instructions or operations: load, store and computation operations, and all the performance tests were carried out on single-precision floating point arrays using a virtual SIMD register of size 4 KB per SPE.

The main parameters in regard to these operations are:

- Load and Store operations move data between the PPE and SPEs using DMA transfers.
- All DMA data transfers are issued by SPEs.
- Load and Store operations can handle unaligned data automatically.
- Load and all other computational operations excluding the reduction operation are non-blocking operations. This implies that once the PPE delivers its message(s) to an SPE, it can then go to the next instruction.
- Store is a blocking operation which implies that the PPE will deliver its message and stall until the requested operation is completed.
- The main activities involved in most of the VSM instruction are: Sending Message from the PPE and processing the request on the SPEs.

All the values and figures in this section are timing the main task of each operation independently excluding any conventional set up such as declarations or initialisations, and all the measurements were taken by the PPE. Table 5.3 presents the average time for loading, storing and performing arithmetic operations of a block of 4096 a single precision floating-point (32-bit) elements on 1,2 and 4 SPEs. Each operation was run 10^6 times using a virtual register of size 4 KB per SPE. The load and store are byte-based operations and therefore data types does not affect the latency time of these memory access operations.

Operation	Time (seconds)		
	1 SPE	2 SPEs	4 SPEs
Unaligned Load	1.581	1.178	0.904
Arithmetic Operations	1.499	1.101	0.833
Aligned Store	4.550	2.401	1.500
Unaligned Store	15.073	7.647	4.036

Table 5.3: The average time of processing a block of 4096 a single precision floating-point (32-bit) elements using a virtual register of size 4 KB per SPE. Each operation was run 10^6 times

Figure 5.25 shows the average latency per a 32-bit word (single precision floating-point). This figure shows that the average latency of load or arithmetic operations per word vary between 0.2 nsec using one SPE to 0.4 nsec when four SPEs were used. These two instructions are non-blocking operations, yet the latency of the load operation, as we can see, is slightly higher than an arithmetic operation. This variation is a result of loading

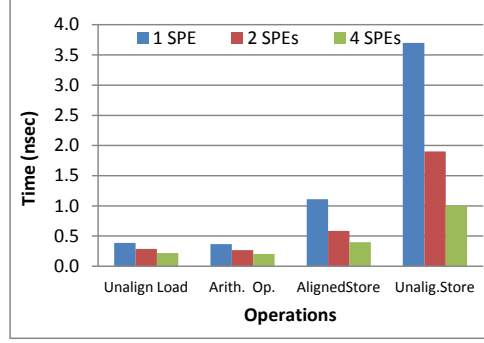


Figure 5.25: The average latency time for processing a 32-bit word (single precision floating-point) using a virtual register of size 4 KB per SPE.

unaligned data. Loading unaligned data does not cost as much as unaligned store because if the data is unaligned, an SPE then needs to load only additional 128 bytes and does not need to use any locks. More discussion on this point can be found in subsection 5.6.3.

The store instruction's design, on the other hand, differs from the load's design. Handling alignment by the SPE on Store is more costly than on Load because it requires three DMA transfers and two locks to handle data synchronization. Also the store operation is a blocking operation which means that the SPEs has to communicate with the PPE using mailbox messages (acknowledgment) once the data has completely left the SPE local memory, and thus no overlapping is allowed as in Load. Moreover, the PPE is required to call the `__lwsync()` function to ensure that data is completely residing in the main memory before executing the next instructions. These factors explain why storing unaligned data costs up to 10 times as much as unaligned load when one SPE is used and up to 5 times (1 nsec) when using 4 SPEs as shown in Figure 5.25. The same figure also shows that the average cost of storing aligned 32-bit word is reduced by a factor of 3.5 as compared to storing unaligned 32-bit word. However, store is not as frequent instruction as load and computation instructions, and thus the cost can relatively be reduced by combining as many operations as possible per an array expression.

Figure 5.25 also points out that blocking instructions, such as Store, are almost fully scalable because the opportunity to overlap communication and data transfers is high, and this can be seen clearly as the store instruction latency was reduced by a factor of $P^{0.94}$, where P is the number of processors. On the contrary, the cost of non-blocking instructions is reduced roughly by a factor of $P^{0.42}$ only where P is the number of processors. non-blocking instructions are not fully scalable because their low-latency does not allow a time window to overlap communication and reduce the cost.

Figure 5.26, which is a graphical presentation of the figures given in Table 5.3, shows that the Load and arithmetic operations have almost the same latency time because they are

5 Virtual SIMD Machine

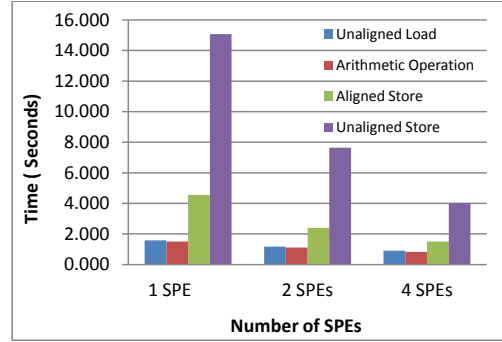


Figure 5.26: The time for processing block of data on 1,2 and 4 SPEs. Each operation was measured using a block of 4096 single precision floating-point values, a virtual register of size 4 KB per SPE and run 10^6 times

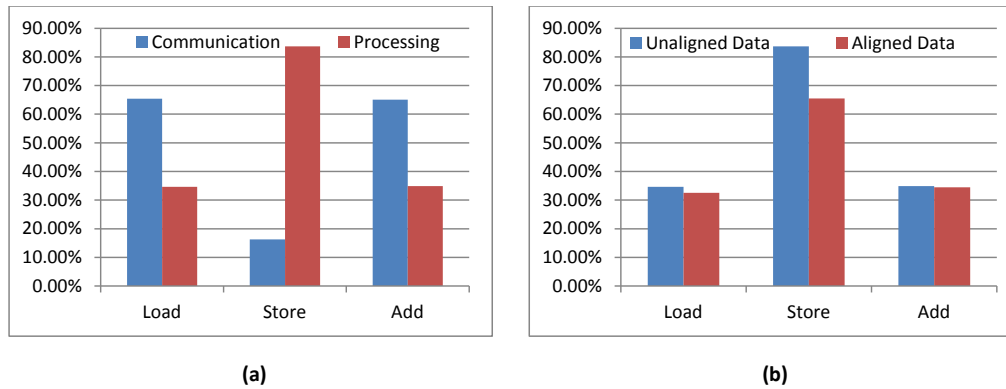


Figure 5.27: (a) The percentage of time spent on the main activities out of the total latency time of each operation (b) The percentage of time spent on processing aligned data as compared to unaligned data.

non-blocking operations which means that the reported costs of these operations represent only the cost to exchange messages between the PPE and SPEs as was explained in Section 5.6.3. Store operation's latency is relatively high even on aligned data because it is a blocking instruction. However, the average cost of each operations decreases of as the number of the SPEs increases. The diagram in Figure 5.26 shows that the time for storing unaligned data using the same above parameters was reduced from around 15 seconds when on SPE was used to about only 4 seconds when 4 SPEs were used. The time reduced by a factor of 3.75 when storing unaligned data and similar speedup was achieved on storing aligned data. The achieved improvement on the Load and arithmetic operations when SPEs were used is relatively small comparing with the Store operation because these is only a narrow window of time in which message exchanges between multiple SPEs can be overlapped.

The charts (a) and (b) in Figure 5.27 represent the percentage of the activities involved in load, store and add operation out of the total latency time of each operation. The activities

involved in each instructions are sending a request (messages) from the PPE to the SPEs and processing the request by the SPEs. Figure 5.27 (a) shows that load and add operations each spent an average of 65% of the total time on communications or sending messages while the store operation spent about 16% on sending a message from the PPE to the SPEs. The variation here is due to the total latency time of the store operation as the percentages were calculated relative to the total latency time of each operation.

Figure 5.27 (b) plots the percentage of the time spent on processing aligned data as compared to processing the same size and type of unaligned data. Let us first comment on the computational operations. The processing time may vary from one computational operation to another, but it should not be effect by the alignment constraints, as we can see in this figure, because all SPE virtual registers are aligned data buffers and consequently all computational operations are performed on aligned data.

Figure 5.27(b) shows that the latency to load unaligned data is slightly higher than loading aligned data. The difference is considerably small because loading unaligned data on an SPE needs to load only 128 bytes additionally and no locks are used. On the store operation, however, we can see a large improvement on aligned data. It drops from around 83% to about 65% of the total latency time of the operation because if data is aligned, the SPE then does not need to bring data from the main memory back and forth as explained in the Store Processing Algorithm presented in subsection 5.6.4.1.

6 Host Compiler Development

The VP front end compiler is already implemented, but it needs very slight changes to handle the machine-dependent features. Also there are several back-end compilers that are already implemented for different architectures but not for the PowerPC architectures[113, 141, 142]. Therefore, it was necessary to develop a sequential back-end compiler to port VP on the Cell's master processor and then extend it in order to collaborate with the VSM model to access the Cell's SPEs. The Cell's host processor belongs to the PowerPC architecture family. PowerPC architectures have specific register conventions and do not have reserved registers such as a stack pointer and a frame pointer. They also do not support a number of basic machine language instructions such as stack PUSH and POP and instructions that handle functions' calling mechanism; like ENTER and LEAVE Intel instructions. These are the main issues that must be resolved explicitly by PowerPC compilers to ensure the compatibility of different modules of a program and to be able to combine these modules as one unit.

VP back-end compilers basically consist of two parts: a targeted machine description and machine-dependent routines. The machine description defines the resources of the target machine such as machine instructions, memory, registers, address modes ... etc, while the other routines solve machine-dependent features such as register conventions and stack operations. The machine description is coded in Intermediate Language for Code Generator (ILCG) notations, and the machine-dependent routines are written in Java. The implementation includes as well a number of assembly macros to handle instructions that are not offered in the PowerPC assembly instruction set such as loading a 32-bit immediate value in a register. The sequential version of the compiler is then extended to collaborate with VSM for parallelising array expressions. The tools that were used in the implementation are GNU tools such as Java compiler and the assembler. The macro processor *m4* was also used as a text replacement tool to unfold machine specification templates.

However, during the PowerPC back-end compiler development, I developed a new approach to optimize the compiler code-generator. This approach is based on genetic algorithm techniques to automatically optimize instructions set ordering instead of the manual

approaches which depend on feedback procedures to enhance the ordering of instructions. The discussion of the GA optimiser, however, is deferred to the next chapter.

This chapter covers the implementation of the standard (sequential) PowerPC back-end compiler and its extended version. The compiler was extended to work together with the VSM model as a single compiler system.

6.1 PowerPC Machine Description

The PowerPC machine instruction set was described using the ILCG notations. The implementation is supposed to include declarations and definitions of available data types, physical machine registers, operators and the instruction patterns. The following are some samples of the basic machine specification.

- Data types declaration

```
type int8=BYTE;
type int16=WORD;
type int32=DWORD;
....
```

- Chain Type definition

```
pattern signed means [int8|int16|int32|int64];
pattern word64 means [int64|uint64];
pattern double means [ieee64];
.....
```

The means portion is used by the unification algorithm to match against an abstract syntax tree node.

- Registers declaration
 - Regular Registers

```
register int64 R3 assembles['r3'];
register int64 R4 assembles['r4'];
...
register int64 R5 assembles['r31'];
```

The assemble component determines what assembly code is to be generated.

– Reserved Registers

```
reserved register int64 R0 assembles['r0'];
reserved register int64 R1 assembles['r1'];
...
```

– Alias Registers

```
/* Stack Pointer (SP) */
alias register word64 SP=R1(0:63) assembles['r1'];
~
/* Frame Pointer (FP) */
alias register word64 FP=R2(0:63) assembles['r2'];
```

Once registers are defined, they should be grouped in sets. For example, the following code illustrates how registers, R0,R1,... Rn, are grouped under the siregs pattern.

```
pattern siregs means[R0|R1|R2|R3|R4|...| Rn];
```

- Referencing Indirect Memory

```
pattern naddrmode (nsrc r) means[ mem(^ (r)) ]
assembles [r];
```

- Operators Definition and Grouping

```
add means + assembles ['add'];
operation gt means > assembles ['gt'];
operation eq means = assembles ['eq'];
pattern oper means [add|mul|dv];
pattern logicoper means [and|or|xor];
pattern cond means [lt|gt|eq|ng|le|ge|ne];
pattern immediate means [int8|int16];
```

- Patterns definition of memory operations

```
pattern EA(reg r) means [^(r)] assembles['('r')'];
pattern lbl(label l) means [l] assembles [l];
```

- Stack Operations

```
/* Increment the SP by the offset i */
instruction pattern INCSP(imm i)

    means[ R1 :=+ (^(R1),i) ]
    assembles['li r1,'i];

/* Push register r on the stack and then update r1(SP) */
instruction pattern PUSHREG(reg r)

    means[PUSH(mainSTACK,^(r))]
    assembles['std ' r ', 0(r1)'
              '\n li r1, -8'];

/* Update r1 (SP)and then pop register r from the stack */
instruction pattern POPREG(reg r)

    means[r:=(int64)POP(mainSTACK)]
    assembles['li r1, 8
              '\n ld 'r', 0(r1)'];
```

- Memory access operation

```

/* Load Instruction */
instruction pattern LOAD(reg r1, reg r2, reg r3)

    means [r1:=(int64)mem(+^(r2),^(r3))]
    assembles ['ldx ' r1 ',,' r2 ',,' r3 ];

/* Store Instruction */
instruction pattern STORE(maddrmode rm, reg r1, word64 t)

    means [ (ref t) rm:= ^(r1) ]

    assembles ['std 'r1',,rm];

```

- Branch Instructions

```

instruction pattern IF(cond c, reg r1, reg r2, lbl l)
    means [if( c((int32)^(r1), (int32)^(r2))) goto l]

    assembles['cmpw ' r1 ',,' r2
              '\n b'c ' ' l];

instruction pattern IFI(lbl l,reg r1,imm s,cond c,int b)

    means[if((b)c((t) ^(r1),const s))goto l]
    assembles['cmpwi ' r1 ',, ' s
              '\n b'c ' ' l];

```

- Arithmetic Operations

```

instruction pattern Op(oper op,reg r1,reg r2,reg r3, int t)
    means[r1 := (t)op( (t)^(r2),(t)^(r3))]

    assembles [op' 'r1',,r2',,r3];

instruction pattern MOD2(reg r)

    means [r:=MOD(^r),2]
    assembles ['andi 'r',,r',1'];

```

- Other Operations


```
instruction pattern LIMM(imm i, reg r)
means [r:=const i]

assembles ['li ' r ',' i ];
```

- Defining Instruction Set

The final step in the machine description is to list all the instructions that will be used to match against the ILCG source code. The following ILCG code shows how the above instructions are ordered in a list named *PPC – Instr*:

```
define(PPC-INSTR, INCP|PUSHREG|...|STORE|IFI|IF|LIMM|OP|MOD2)
```

All the instructions and operation patterns must be defined at the end of the machine description file.

6.2 Machine-dependent Routines

The automatically produced methods, *PPC.java*, was extended in a new class, called *PPCCG*. The extension includes the following methods.

- *cgApply()*:

This routine's task, in general, is to arrange arguments passing based on the PowerPC ABI and generate corresponding ILCG notations. It receives a node of type procedure, and it starts by getting a rough estimate of the number of registers needed to pass the arguments of that procedure, and based on this information, it then takes the following steps to generate the required ILCG code:

- Determine the number of registers needed to pass the arguments. This must adhere to the PowerPC ABI shown 3.4.
- Put the first parameters into the argument registers and reserve the used registers.
- If the available registers are not enough to hold all the parameters, it then calls the *cgPushit()* routine to push the remaining arguments on the stack and updates the stack pointer.

- *cgPushit()*

It generates the required code to allocate parameters on the stack. It receives a node which needs to go on stack and returns an offset from the stack pointer. The main steps in this routine are:

- Gets the length of the node in bytes
- Casts the node to a reference as it will be located on the stack.
- Reserve the proper space on the stack while considering any alignment constraints when updating the offset.

- *cgReturn()*

This method determines the proper register to be used to return values. PowerPC designates different registers for different data types, and thus the routine has to first determine the data type to be returned, and it then determines the register.

- *cgProcedure()*

This is a focal function that examines the type of a given procedure, establishes a procedure signature and calls the proper routines to generate assembly code that corresponds to the source code. This routine takes two arguments: an object of type procedure and another one of type Walker. The Walker class provides generic utility methods that can be used by code generators, and if the code generation process is succeeded, it then returns true. The main steps in this routine are:

- Determine if the procedure is an external (imported) by adding the proper assembly directive.
- Plant the directive to identify the start of the procedure.
- Generate a label to the procedure and add it to the code.
- Call the ENTER() function and pass to it the lexical level of the procedure to establish a new execution environment for the called procedure.
- Call the code generator routine on the class Walker. The routine walks through the syntax tree and matches it against the machine description. This method generates a stream of machine code instructions that correspond to the source code in the body of the procedure.
- Calculate and set the size of the stack frame needed.
- Call the LEAVE() function to produce the code for exiting a pascal procedure safely.

6.3 Assembly Macros

Besides the Java machine-dependent routines, assembly macros were implemented to handle undefined operations and functions such as loading 32-bit and 64-bit immediate values (constants), trigonometric functions and data alignment. The following discussion looks at three samples from the implemented macros.

- Loading 32-bit Immediate Value

We had to implement this assembly macro to allow loading a 32-bit immediate value in a register because PowerPC architectures offer only 16-bit immediate instructions.

```
.macro loadintr value,reg
.section .data

    1: .long \value

.section .text

    lis 27,1b@ha
    addi 27,1b@l(27)
    lwa \reg,0(27)

.endm
```

The first line defines the macro's name and the parameters it takes. It is called `loadIntr` and takes two parameters (`value` and `reg`). It can be used to load a 32-bit integer value into a register. Then we have in the second line the directive `.data` which indicated that what follows is data not code. The assembly statement in the data section allocates a 32-bit (long) word in memory (labeled with `1:`) and sets it with the constant value. The label `1` is a local symbol that can be used to reference that memory location. The directive `.text` signals to the assembler that what follows is code not data. The first two assembly instructions load the address of the location `1`. That is, the instruction `lis` loads the top 16-bit of the location `1` into register number 27 and shifts them to the left, and the instruction `addi` (load immediate) loads the lower 16-bits into the same register number 27. The symbol `1b` in the `1b@ha` and `1b@l` symbols means the most recent defined (`b` for backward) label of the number `1` while `@ha` and `@l` means the higher and lower 16-bit respectively. The last instruction `lwa` in the code section loads a 32-bit integer word starting from the address into register number 27 into the given register `reg`. The `.endm` directive ends the macro.

- Trigonometric Functions

The following macro shows an example of how the trigonometric functions are handled using assembly code.

```
.macro fsins f
.section .text

    fmr 1,\f
    bl sin

.endm
```

This macro takes one argument; it is a floating-point register *f*. The instruction `fmr 1, \f` then moves the contents of register *f* into floating-point register number FPR1 as it will be, according to the PowerPC ABI, passed to the followed instruction `bl sin` which calls function `sine`.

- Align Load Instructions

Because the PowerPC Architectures use instructions that are 32-bit word-aligned.

```
.macro unalignload reg base offset
.section .text
.if ( \offset & 3) != 0

    li 30,\offset
    andi. 30,30,3
    neg 30,30
    addi 30,30,4
    addi 30,30,\offset
    add 30,30,\base
    lwz \reg,0(30)

.else

    lwz \reg,\offset(\base)

.endif .endm
```

Action	Intel	PowerPC
1- Pushing the frame pointer on the stack	push(EBP)	stw r31,0(r1)
		addi r1, r1, -4
2- Maintain a copy of the SP;	Temp \leftarrow ESP	la r30,0(r1)
3- Check nested level (NL)	IF (NL > 0)	L1: cmpwi NL,0
Copy previous frame pointers in display area	FOR i \leftarrow to(NL-1)	ble L2:
3.1 move FP to previous frame location	EBP \leftarrow EBP - 4	addi r31,r31,-4
3.2 Store previous nested FP in display area	push(EBP)	stw r31,0(r1)
		addi r1, r1, -4
3.3 Update the level		addi NL,-1
3.3 Go back to step 3.1	ENDFOR	b L1:
4- Store the original SP onto the stack	push(Temp)	L2: stw r30,0(r1)
	ENDIF	
5- New FP points to current SP (r30 in step2)	EBP \leftarrow Temp	mr r31, r30
6- Computes new frame size FS & update SP	ESP \leftarrow EBP - FS	addi r1,-FS(r31)

Table 6.1: Emulating Intel ENTER instruction on PowerPC

6.4 Stack Frame Operations

Since PowerPC architectures do not support Enter (prolog) and Leave (Epilog) instructions, it was necessary to overload these methods as members of the class *PPCCG*.

- *ENTER()*

This routine creates a stack frame, updates the SP, updates the frame pointer FP, and links nested procedure if there is any. It must be called whenever a procedure or function is called. Architectures, such as Intel, offer hardware support for such an operation but not the PowerPC. Table 6.1 presents an emulation of the Intel ENTER instruction for the PowerPC in assembly language.

The PowerPC assembly instructions in the third column of Figure 6.1 are what the implementation of ENTER() function is expected to generate.

- *LEAVE()*

The following Java code shows the implementation of the function leave() which releases an active stack frame and updates variables such as the return address.

```

private void leave(Walker w) {
    w.buf.writeln("Ebilog");

    /* load return address in Reg0 */
    w.buf.writeln("ld r0, 16(r1)");
    w.buf.writeln("mtlr r0");
    /* Update the SP to point to the previous frame */
    w.buf.writeln("addi r1,r1,frameSize");
    /* branch to the location in the link register */
    w.buf.writeln("blr");

}

```

6.5 Compiler Building Process

This section documents the main steps involved in the development process of the PowerPC back-end compiler along with the tools used in each step:

- Define the target machine registers, instruction patterns, and operations and save them into a macro file called *PPC.m4*.
- Transforming the *PPC.m4* into *PPC.ilcg* format using the *m4* macro processor.

```
m4 PPC.m4 PPC.ilcg
```

- Translate the ILCG code *PPC.ilcg* into Java methods using the ILCG code-generator generator, and keep the generated code in the file *PPC.java*.

```
ilcg.ILCG PPC.ilcg PPC.java
```

- Compile the generated Java methods to generate class for each method.

```
javac PPC.java
```

- Compile the machine-dependent routines in the class *PPCCG*.

```
javac PPCCG.java
```

- Aggregate all the produced classes including the front end classes into one Java archive file. This is the last step in the compiler building process.

6.6 PowerPC Compiler Extension

This section discusses the machine specification extension development. The work involved extending the machine description of the PowerPC back end compiler to be capable of collaborating with the VSM model. This includes defining a virtual SIMD register set and introducing a new instruction set that operates on these virtual registers. I also had to modify some machine-dependent routines such as ENTER and LEAVE to create and terminate threads.

6.6.1 Packed Virtual SIMD Registers

The virtual vector (large) registers are designed to be used by virtual SIMD instructions which imitate SIMD instructions on the SPEs. The length (VECLLEN) of these packed registers can vary, and I experimented with register lengths between 1024 and 16834 bytes. The register file in this extended compiler consists of 8 vector registers, labelled NV_0 to NV_7 . The following ILCG code, which declares the single-precision floating-point NV register set, illustrates both the machine semantics and the specification system:

```
/* Machine Description of the Virtual SIMD Machine Registers */
define(VECLLEN,1024) /* this is an experimental parameter */
register ieee32 vector(VECLLEN) NV0 assembles[' 0'];
register ieee32 vector(VECLLEN) NV1 assembles[' 1'];
register ieee32 vector(VECLLEN) NV2 assembles[' 2'];
register ieee32 vector(VECLLEN) NV3 assembles[' 3'];
register ieee32 vector(VECLLEN) NV4 assembles[' 4'];
register ieee32 vector(VECLLEN) NV5 assembles[' 5'];
register ieee32 vector(VECLLEN) NV6 assembles[' 6'];
register ieee32 vector(VECLLEN) NV7 assembles[' 7'];
/* Groups vector registers under type nreg */
pattern nreg means[NV0|NV1|NV2|NV3|NV4|NV5|NV6|NV7];
```

The above ILCG defines each virtual register NV_i as a 1024-word vector register. These registers can be used to generate a set of RISC like register load, operate and store instructions. A packed virtual register can be processed by one SPE or multiple SPEs. If a single SPE is used, the entire contents of each virtual register (NV) is copied into an aligned buffer on that SPE. If multiple SPEs are used, each virtual register (NV) is then equally distributed on the available SPEs. For example, if two or four SPEs are used, then each SPE holds a half or a quarter of the register respectively. The partitioning process is

the responsibility of the PPE stub routines which must compute the starting addresses and broadcast the VSM instruction to each active SPE. Then each SPE can process its portion of the register in parallel with the other SPEs. To illustrate how the starting addresses are computed, we take the following simple example. Suppose that the virtual registers can hold 1024 floats point values and 4 SPEs are used; that is, each SPE operates on 256 elements. Now, if the starting address to be accessed in the main memory is 0XA0000, the responsible PPE routine then should broadcast to the four SPEs the following starting addresses:

```
000A0000 →SPE0 ; Start address of the vector
000A0400 →SPE1 ; offset of 256 floating points from start
000A0800 →SPE2
000A0C00 →SPE3
```

In regard to the number of registers, to evaluate an array expression within registers, the maximum number of registers the evaluation may require would be equal to the number of operands in the expression, and thus in the worst case scenario the compiler can evaluate an array expression comprising of 8 operands. However, this is often not the case because after each intermediate primitive binary operations, as in VP, one of the registers can be reused. To illustrate that let us take the following simple example which shows how to approximate the number of registers required for the evaluation of general expressions with binary operation:

Consider the arithmetic expression

$$A := B * (C - D) + Z / Q$$

where A, B, C and D are vectors. In order to evaluate this expression, the compiler should generate code such as

```
R0 ← Z
R1 ← Q
R0 ← R0 / R1
R1 ← C
R2 ← D
R1 ← R1 - R2
R2 ← B
R1 ← R1 * R2
R0 ← R1 + R0
R0 → A
```


The above example shows that an expression with 4 operators and 5 operands requires a maximum of 3 registers. Ershov in 1958 proposed an approach that can be used to find an optimal number of registers that are required for the evaluation of an expression of primitive binary operators [157]. Ershov suggested that if an expression contains n number of binary operators and $n + 1$ number of operands, then the number of registers $NReg$ needed for the computation is $NReg \leq \log_2(n + 1)$ [157, 158]. Accordingly, a set of 8 virtual registers would be enough to evaluate one very long expression at a time.

A virtual register length obviously has an influence on how an array expression is handled if the length of the arrays being operated on in the source code are bigger than the virtual registers. In fact, this is one of the reasons which motivated us to choose Vector Pascal because its front end compiler is already designed to handle such situations when dealing with SIMD instruction sets. If the length of the arrays are an integer multiple of the register size, the front end compiler can then easily unroll the entire expression into multiple successive calculations on sub-arrays of length equal to the virtual register length. For example, if an array expression contains arrays of size 2048 and the virtual registers are of length 1024, the compiler would then unroll the expression into two successive calculations. However, if the array size is bigger than the virtual SIMD register, the compiler should generate virtual SIMD instructions which handle the portion of the array, which fits in one or multiple virtual registers, and should also generate PPE scalar instructions to handle the remainder.

6.6.2 Virtual SIMD Instruction Set

As was mentioned in the previous chapter, the VSM offers the PPE stub routines or functions with two arguments to support primitive operations. The main task of most PPE routines is to dispatch a request to the SPE(s) to perform a given operation. Thus, the idea here is to introduce new machine instructions; called Virtual SIMD Instructions (VSIs). VSIs are of two address register to register formats. They are a set of RISC like register load, operate, store operations. The VSI set supports basic operations, such as Load, Store, Add, Sub, Sqrt ...etc, in a mapped fashion. The compiler's code generator should look at the VSIs as a set of SIMD like instruction set with a high degree of parallelism. VSIs can deploy the proper information in the right registers and invoke the right PPE sub routine. The role of the VSIs ends at this point because it is the responsibility of the PPE interpreter to communicate with the SPE(s) and to handle other parallelisation issues such as data partitioning and synchronisation.

Semantics of some samples of the Virtual SIMD Instructions and how they were mapped into machine code are given in the following subsections.

6.6.2.1 Virtual SIMD Load and Store Instructions

The virtual SIMD load and store instructions can be used to move data within the Cell processor. From the PPE interpreter point of view, these operations are carried out using underlying DMA operations. A DMA transfer, as was mentioned in the introduction of the Cell, is basically a function with several parameters, yet the most important parameters are: local store address, main memory address. From the code generator point of view, these virtual SIMD instructions are basically required to call the right PPE stub routine providing that it sets the required information in the proper registers. Consequently, the implementation of these instructions must adhere to the PowerPC ABI shown in 3.4 by using the registers *r3* and *r4* to pass this information. The first register should be loaded with an integer *n* which determines the virtual register's number to be used. The second register should be loaded with the starting address of the data to be moved from/to the main memory. This implies that the implementation of each instruction requires two assemble instructions to set these two parameters and a third assembly instruction to call a PPE function for the particular operation. The following ILCG code shows the Load and Store instructions:

```
instruction pattern LOADFLT( naddrmode rm, nreg n)

means[ n :=(ieee32 vector(VECLLEN))^ (rm)]
assembles['li 3, ' n
          '\n la 4,0(' rm ')',
          '\n bl LoadVec'];

instruction pattern STOREFLT( naddrmode rm, nreg n)

means[(ref ieee32 vector(VECLLEN)) rm := ^(n)]
assembles['li 3, ' n
          '\n la 4,0(' rm ')',
          '\n bl StoreVec'];
```

Notice here that *n* is a variable of type *nreg* which was defined above. The code generator will select the register (*n*) to be used from the registers (*NV₀ - NV₇*) listed under the *nreg*. The selection depends on the available registers. The LoadVec and StoreVec are the PPE

functions which are supposed to handle the load and store of floating point vector of size VECLLEN.

DMA transfers also require the size of the data to be transferred. This task is left to the PPE functions to determine based on the data type.

6.6.2.2 Virtual SIMD Computation Instructions

Computations operations could be classified based on the number of operands (arity) that an operation takes. There are binary operations, such as add, multiply and replicate, and unary operations such as sqrt, sin and cos operations. To set the environment for executing any operation on the SPE, computation virtual SIMD instructions first loads the required information, which determines the operands involved in the operation, in the proper registers based on the PowerPC ABI shown in 3.4, and then calls the right PPE stub routine. For example, a binary operation, such as Add, requires two integer values which represent the numbers of the two virtual vector registers that it operates on. In this case, the machine physical registers, r3 and r4 can be used to pass these parameters. The operands in other binary instruction are sometimes of different register types. For example, the replicate instruction operates on a scalar value and a vector register in which the scalar will be replicated. Thus the scalar needs a physical machine register while the second operand needs a virtual SIMD register. However, in unary instructions, the virtual SIMD register, which is the only operand, is used as a source and a destination.

- Instructions of Binary Operations

We present here two examples of binary operations which were implemented differently. In the first example, the operation involves two arrays, and thus the PPE function requires two parameters; say integer n and m, to determine the virtual vector registers to operate on. Accordingly the ADDFLT instruction, given as an example here, must load n and m into the PPE general purpose registers, r3 and r4 respectively before it calls the PPE function AddVec. The implementation of the ADDFLT instruction is shown in the following ILCG code:

```
instruction pattern ADDFLT(nreg n,nreg m )
means[n:= +(^n),^(m)]

assembles['li r3, ' n
          '\n li r4, ' m
          '\n bl AddVec'];
```

The first line defines the instruction ADDFLT that takes two parameters *n* and *m* of type *nreg*. The *nreg* is a pattern which represents, as was defined above, a group of virtual SIMD registers. The second line specifies the semantics of the instruction. The semantic of this operations specifies two sources register numbers are *n* and *m*, the operation (+), and where the result should goes; into register *n*. Note the ^ operator means “get the contents of “. The last 3 lines show the standard assembly code that is supposed to be generated. The first two assembly instructions load the PPE general purpose registers; *r3* and *r4*, with the virtual register number *n* and *m*. The third assembly instruction is a branch instruction that leads to a call to the PPE function *AddVec*.

The other example of binary operations differs from the previous example in the type of argument to be passed. The following ILCG code:

```
instruction pattern REPFLT( nreg n, sfreg r)
means[ r:=rep(^n),VECLLEN]
assembles['li 3, ' n
          '\n fmr f1,' r
          '\n bl RepVec'];
```

shows how the replicate floating-point instruction RELFLT is implemented. This instruction does not operate on two vector registers like the ADDFLT operation, instead it replicates a floating-point scalar into a vector register. This implementation shows that the type of the first argument is the same type used in the previous examples, but the type of the second argument is different. The type used in this instruction refers to a singed floating-point machine register (*sfreg*) which is supposed to hold the scalar value. Consequently there is a need here for an assembly instruction such as *fmr* to move a scalar value into the first floating-point machine register (*f1*) in order to pass the floating-point value.

- Instructions of Unary Operation

We present here the implementation. This example shows the following ILCG code of a unary operation:

```

instruction pattern SQRFTFLT(nreg n)
means [n:=SQRT(^n)]

assembles['li 3,' n
          '\n bl SqrtVec'];

```

The first line defines the instruction SQRFTFLT which takes one parameter, and the second line specifies the semantics of the instruction in which the source register *n* is also used as a destination to hold the result of the square root operation. The third line includes two standard assembly instructions. The first instruction (`'li 3,' n`) loads the PPE general purpose register 3 with the virtual register number *n* while the second (`bl SqrtVec`) is a branch instruction which leads to a call to the PPE stub routine called `SqrtVec`.

6.7 Building the VP-Cell Compiler System

This is the third stage in the development of the VP-Cell compiler system. It involved the integration of the VSM interface with the VP compiler as a single compiler system that has the capability to automatically parallelise and execute array operations on the SPEs. This last phase included extending the PowerPC back-end compiler to incorporate a user-level instruction set (Virtual SIMD Instructions) using a standard instruction set. The extension included a definition of a set of 8 virtual SIMD registers or vector registers and a description of the VSM instructions. The ILCG file, which contains the description of these resources, was appended to an existing ILCG description of the PPE to give a complete description of an integrated machine. The ILCG compiler then builds a code generator in Java which is compiled and linked with the Pascal compiler. The extension included also a slight change in the code generator routine which determines the degree of parallelism the target machine can support on different data types. It is important to note that no changes were necessary in the parser or high level optimizer of the compiler itself in order to achieve this. By supplying the machine code generator with the size of available virtual or vector registers, the parallelisation processor is simply piggybacked on the way the compiler front end decomposes array operations for whatever processor it is targeting. The code generator 'thinks' it is generating a set of RISC like register load, operate and store instructions. These actually expand out into calls to the PPE stub routines which pass the instruction arguments as messages to each active SPE using its mailbox registers.

The ILCG file describing the VSM is appended to an existing ILCG description of the PPE to give a complete description of an integrated machine. The ILCG compiler then builds a

6 Host Compiler Development

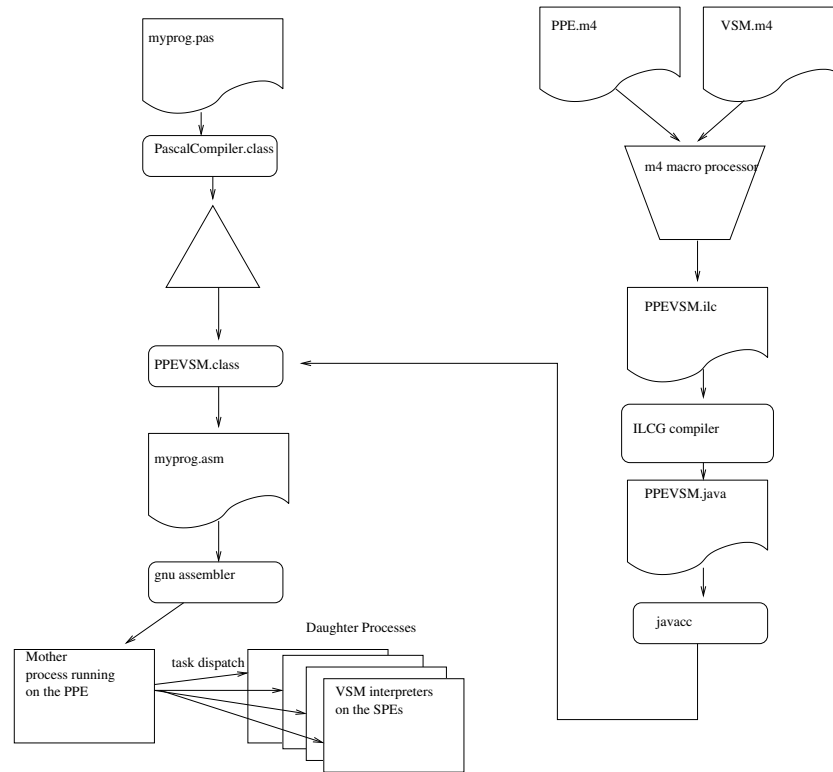


Figure 6.1: Building the code generator and compiling a Pascal program

code generator in Java which is compiled and linked with the Pascal compiler; the process is shown in Figure 6.1.

We illustrate how VP-Cell handles the execution of array expressions with the following simple Vector Pascal program which reads two arrays of reals from standard input and prints the average of the two arrays given that the VSM register size is 8192 bytes.

```
01 program Av(input,output);
02 const arraylen=4096;
03 type vector=array[1..arraylen] of real;
04 var a,b,average:vector;
05 begin
06     readln(a);readln(b);
07     average:= (a+b)/2;
08     writeln(average);
09 end.
```

When the compiler processes line 7 it sees that it has to perform calculations on vectors 4096 elements long, so it asks the code generator what is the maximum degree of parallelism this machine can do on reals. On a simple scalar CPU like the old Pentium the

answer would be 1, and the compiler would generate a loop going round 4096 times to do the calculation. For the VSM though, the answer on the degree of parallelism comes back as 2048, so the compiler decides that it can unroll the entire task into two successive calculations on sub-arrays of length 2048. It thus schedules loads of the two halves of a and b into a pair of NV registers followed by add and divide instructions working on the entire vector registers.

The program takes the form of a mother process on the PPE and 1, 2 or 4 daughter processes on the SPEs. At the start of a program, one or more copies of a VSM interpreter program are initiated, one per SPE. A compile time flag can be used to select how many SPEs are to be used.

If a single SPE is used, the entire length of each NV register, which is in this example 2048 real elements, is mapped into the local memory of that SPE. The interpreter sits in a loop reading instructions from the mailbox and dispatching interpreter tasks corresponding to the instructions. If two or four SPEs are used, each SPE operates on 1024 or 512 elements respectively. In this case, the PPE stub routines broadcast the VSM instruction to each active SPE. Because each SPE can now process its portion of the VSM register in parallel with the others, adding more SPEs accelerates the execution process.

6.8 Coding

The following points perceive what has been implemented and how many lines of code were produced

- The PowerPC machine description file, PPC.ilcg, consists of more than 1250 lines of ILCG code written to define around 209 abstract instructions and to map these instructions into PowerPC assembly instructions. This file then fed to the code-generator generator which automatically generates Java methods for each described operation. The generated methods are kept in a PPC.java file that contains more than 47000 lines of automatically generated code.
- The machine-dependent routines are implemented in Java as an extension to the automatically-generated program. The program extension is called PPCCG.java, and it handles stack operations and functions calling mechanism. This program contains around 750 lines of code.
- The macros were written in Gas assembly language. The macros file contains about 230 lines of assembly instructions.

7 Code Generator Optimiser

There are a variety of problems which could be optimized automatically in the compiler domains, but the intention here is to focus on using genetic algorithms for optimising code generators.

Code generators of some programming languages such as Prolog and VP employ a unification based technique for matching or constructing logical proofs [42, 159]. For example, VP code generators employ a pattern unification matching technique to map semantics into assembly code. Adjusting code generators manually usually depends on feedback procedures to enhance the ordering of instructions, but this tuning process is very time consuming. The manual approach also requires a considerable human effort and does not guarantee that a given sequence of instructions is a good choice. Thus, it will be productive to have a tool that can assist compiler designers especially for newly developed compilers in tuning and enhancing code generators that consequently reduces execution time for a given application. In this project, I developed a new approach to optimise a compiler code generator, and the novelty of this approach is using Genetic Algorithm (GA) techniques to automatically optimise machine instruction ordering.

This chapter discusses in detail the code generator optimiser. It starts with an introduction of generated code problems which could be optimised using GAs. It then looks at two existing approach in which GAs used to optimise code. After that, it gives a brief description of the basic algorithm which developed to optimise compiler code generators and looks at the design and implementation aspects. It concludes with experimental evidence that shows the use of such genetic algorithms can improve the quality of automatically constructed code generators.

7.1 Introduction

Some compilers use unification to match generated-tree nodes with the semantics of the target machine and generate the assembly instruction (s) associated with the first matched pattern. This means that there is a possibility that a single node will match a number of

semantics, and hence the order of instructions patterns can affect which of several possible matches will succeed. Using unification algorithms raises the problem of choosing the optimal instructions order from a vast range of semantically equivalent code sequences.

Thus, this work investigated whether genetic algorithm techniques can help in getting a better instructions order that consequently reduces execution time. This piece of work was carried out in the early stages of the PowerPC back-end compiler development in an attempt to improve its code generator because it was newly developed and had not gone through many hand ordering optimisations. Actually, during the verification and testing of the compiler implementation, we notice on different occasions that the number of generated instructions to achieve a certain semantic effect could be reduced by just reordering the machine instructions set.

However, this approach differs from two earlier approaches, which shall be introduced shortly, in the targeted code. The previous two approaches optimise generated code or a single program at a time whereas our optimiser targets code generators. By optimising the code generator itself, the optimiser is run only during the compiler development to improve its code generator and obtain speedups in many applications subsequently translated by the optimised compiler. However, reducing the generated code is highly expected to result in less execution time, yet this is hardly proven due to the lack of reliable instruction times.

7.2 Previous Work

GAs could also be used to optimise generated code. The size of generated code for embedded systems is a critical issue, and software developers often tolerate much longer compile time in the hope of reducing the size of generated code. There are two approaches that have previously been proposed to use GAs for optimizing generated code.

The first approach used GAs to optimize restructuring FORTRAN programs for SPMD running on parallel machines, and it was called the Genetic Algorithm parallelisation System (GAPS) [160]. This technique was used for optimizing the global overhead of parallelising loop-based FORTRAN applications. This form of parallelisation usually requires reconstructing and transformations of original code, such as loop fission and loop fusion, and results in an infinite number of transformations. The GAPS technique was an alternative to the conventional transformations process that is basically based on mapping each statement in the source code into a sequence of alterations. GAPS was also proposed as an enhancement of other existing approaches which attempt to optimize individual statement overheads but not the global overhead of transforming an application or a given program [160].

The second proposal used GAs for optimizing the dual instruction set of the ARM processor. The ARM processor is heavily used in embedded machines. In addition to its standard RISC instruction set, ARM supports a reduced instructions set, called Thumb [161]. The Thumb instructions have smaller instruction lengths than the original instruction set. A program compiled using only Thumb instructions uses more instructions than the same program compiled using a standard instructions set, and it is consequently slower [161]. Because the dual instruction sets could affect the efficiency of compiled programs in terms of performance and space, this approach helps a code generator to swap between the two instruction sets in order to optimize a program's execution time and its code size.

However, genetic algorithms could also be used to optimise the construction of code generators that are based on unification algorithms. Unification is the process of unifying different representations in an attempt to resolve the satisfiability problem. Different programming languages use unification algorithms for different purposes. For example, the Haskell programming language uses unification for the type inference problem while Prolog and Vector Pascal depend on unification algorithms for pattern matching.

7.3 Permutation Technique

Permutation is rearranging a given finite set of objects or values, and it can be a useful encoding technique for some ordering problems [130, 162]. The most common example of a permutation problem is the travelling salesman problem. A solution of the classic travelling salesman is basically a list of cities in which each city must occur once and only once. A simple way to represent a solution of C cities is a list in which each city is given a unique integer number, say $0 \dots C$. Accordingly, any a proposal (solution) or a map of the cities for a salesman must include all the integers from 0 to C in no particular order. From the salesperson's point of view, the best solutions (order) is the shortest paths to visit all the cities, but the search space will be very large to span even with a small number of cities. This representation raises the problem of finding the permutations of these integers (cities), in which each integer is presented only once, that deliver optimal solutions for a sales person. For this reason, there have been past studies on how to use Genetic Algorithms to encode and solve travelling salesman problems [162].

The problem of instruction ordering in compilers is also a permutation problem like the classic travelling salesman problem. In compiler development, the most important part of code generators design is the instruction sets which normally go through successive enhancement and modifications to improve the quality of constructed code generators. This lineage production can be viewed as an inherited genome, and therefore genetic algorithm techniques would be a promising code generation strategy that can help in finding

```

instruction pattern IFI(lbl l,reg r1,immupto16 s,cond c,int b)
    means [if((b)c((t) ^ (r1),const s))goto l]
    assembles ['cmpwi ' r1 ', ' s
              '\n b'c ' ' l];

instruction pattern IF(cond c, reg r1, reg r2, reg r, lbl l)

    means [if ((int32) c((int32)^(r1), (int32)^(r2)) ) goto l]
    assembles ['cmpw ' r1 ', ' r2
              '\n b'c ' ' l];

instruction pattern LIMM(imm i, reg r)

    means [r:=const i]
    assembles ['li ' r ', ' i ];

```

Figure 7.1: Branch Instructions Patterns in ILCG Using Gas Assembly

instruction schedules that deliver near-optimal performance for a particular machine. Permutation techniques can be useful in construction compilers code generators, especially those employing unification algorithms for matching machine semantics (patterns) with intermediate forms of source programs. Where multiple alternative patterns are possible unification algorithms basically output the first pattern whose matching succeeds. As a result, the efficiency of this matching process output is partially subjective to the order of the instruction patterns.

7.4 Why Instructions Ordering is a Problem

Instructions ordering in unification based compilers is a problem that has two faces: productivity or performance and efficiency. Compiler code generators, such as VP, employ unification algorithms to match intermediate representation of source code with machine semantics. They are basically supplied with a machine instruction set as a list which contains all the machine patterns in any given order. This list will be used to match against the intermediate (ILCG) code of source programs.

VP compilers use unification to match machine semantics with the abstract syntax tree of translated code and consequently generate assembly code. However, if multiple alternative patterns are possible, the matching process then will raise the problem of choosing the optimal instructions order which could be viewed as one face of the instructions ordering problem. This optimisation process is often based on manual feedback procedures to

```
li r5,0
cmpw r4,r5
```

Figure 7.2: Assemble Code

enhance the machine code generators, yet the manual tune up often requires a considerable human effort and does not guarantee that a given sequence of instructions is a good choice.

The order of the instructions in the instruction set (list) is a crucial issue as there should be several ways to match generated-tree nodes with a given intermediate code. To illustrate that, let us take a simple example of boundary checking. First assume that the target machine description includes the instructions patterns; IFI, IF and LImm, as shown in Figure 7.1, as representations of the assembly `cmpwi` and `cmpw` instructions.

The IFI instruction in Figure 7.1 basically compares a register with a 16-bit immediate value while the second instruction uses two registers to carry out the comparison. The LImm instruction represents a assembly instruction (`li`) that loads a 16-bits value in a register. Though, these three instructions could be listed in $n!$ orders, we only consider two different orders that could lead the compiler to generate different instructions. In these two orders we shall swap only the instructions IFI and IF as this will be enough to illustrate how the instructions order may effect the performance. Now, to check the lower boundary of an array, say 0, giving these instructions, a compiler code generator would have at least two alternatives to perform the checking. The first option would be if the instructions were listed in the instruction set (PPC-INSTR) in the following order:

```
define (PPC-INSTR,...|...|IF|LImm|IFI|...|...)
```

The code generator then most likely will use the register to register compare instruction because it sees the instruction IF before the instruction IFI. Thus, the compiler will attempt to use the IF (`cmpw`) instruction if it can place the constant value (0) into a register. This condition can be easily fulfilled by the following instruction which loads an immediate value (0) in the register. Under these circumstances the code generator will conduct the comparison using two assemble instructions as shown in Figure 7.2. The generated code assumes that register (r4) holds the array subscript and register (r5) holds the lower bound.

The alternative is to order these three instructions as follows:

```
cmpwi r4,0
```

Figure 7.3: Optimal Assembly Code

```
define (PPC-INSTR,...|...|IFI|LIMM|IF|...|...)
```

The code generator in this case sees the IFI instruction before the LIMM and IF instructions, and therefore it will choose the IFI instruction pattern which results in conducting the comparison using only one assembly instruction; see Figure 7.3. This generated instruction does exactly the same task that the two instructions given in Figure 7.2 do.

Note that as a result of the second instructions order the number of the generated assembly instructions for checking the array lower bound were reduced from two to only one instruction. This could be an optimal order in terms of the number of instructions being reduced to half, yet it does not mean that the total latency is also reduced by the same factor because this depends mainly on the cost of the reduced instructions (the alternative solution) as compared to the remaining instructions. Thus, one can infer here that the order could affect the code generation process and programs execution time.

The size of the search space to be explored and the efficiency in finding the appropriate solution can be seen as the other face of the problem. Though choosing the proper ordering to improve performance can be done manually, the manual approach is less efficient than automatic techniques because the size of the search space to be spanned would be very large and thus there might be better solutions than the hand ordering one. Actually, finding near-optimal or better solutions is even a challenging process at the first stages of the compiler development due to the problem size. A machine with N instructions means there are $N!$ possible orders in which the instructions could be listed. For example, on a machine with only 100 instruction patterns, a compiler developer of languages, like Prolog and VP, are confronted with a massive search space that has around 10^{157} combinations of instruction orders, and this number grows very rapidly even for a few additional instructions.

In view of these considerations, we developed an optimiser using genetic algorithm techniques to optimise machine instructions ordering automatically in the hope of helping to construct compiler code generators .

7.5 Genetic Algorithm Approach to the Problem

The instruction ordering problem is also a permutation problem similar to the travelling salesman problem as the solutions in both cases can be represented as lists of objects. Thus, we present here a genetic algorithm that is based on a new permutation technique for solving the instructions ordering problem. Our genetic algorithm basically starts with a genetic pool of different instruction pattern (opcodes) orders that potentially includes one solution as a default order which could be the existing hand ordering version for existing code generators. It then evaluates these solutions and selects the fitters to be used for reproducing offspring. After that, we perform genetic recombination between a pair of fittest solutions, and re-evaluate the new solutions in order to reselect and reproduce offspring of next generations. The algorithm 7.1 demonstrates these main steps in solving the instruction ordering problem using genetic algorithms.

Algorithm 7.1 Basic Genetic Algorithm for Instruction Ordering

// Create Initial Population

```
{ Get default machine instructions order (first genome);}
{ Generate random populations (excluding first & last genomes);}
{ Reverse the default instructions order (last genome);}
```

// Evaluate the Fitness of the First Population

```
{Fitness function;}
```

// Generate and Evaluate New Generations

```
While ( Number of Generations not Satisfied)
{
```

 // Select Parents for Reproduction

```
    {Select fitter solutions;}
```

 // Generate Offspring Using Selected Pairs

```
    { Perform a single-point crossover;}
    { Perform a flip-bit mutation;}
```

 // Evaluate the Fitness of Successive Populations

```
    {Fitness function;}
```

```
}
```

7.6 Key Design Aspects

7.6.1 Environmental Variables

In order to start the optimiser, users are required to provide some information or parameters. Some of these parameters have default values, yet some need to be supplied by the user. These parameters should be prefixed with the symbol “-” and the letter(s) denoted for the individual parameters as shown below.

1. Number of populations
2. Number of solutions in a population
3. Number of offspring
4. Names of training programs
5. Targeted machine’s name.

7.6.2 Representing Solutions

There are three levels of representations:

1. A solution is a list of N instruction patterns (opcodes).
2. A permutation list L that is made of integers in the range $0 \dots N - 1$ in which each number must occur once.
3. Genomes are encoded as binary bitstrings each of length $N - 1$. A genome G is a set of parameters that defines a proposed solution to the problem, but it does not encode the permutation. In other words, a genome is like a program for a permutation machine.

7.6.3 Encoding first population

The first step that the optimiser does after being supplied with the parameters is to generate an initial population ($j = 1$) of genomes $G_{1..S}$. This step is carried out only once to generate the first population, and it involves:

1. Setting all the bits of the first genome G_1 to 0 (Null permutation). The permuted list that is generated based on (G_1) is supposed to be identical to the default opcodes order.
2. Setting the bits of $S - 2$ genomes randomly (Random permutations).
3. Setting all the bits of G_S to 1 (Full permutation). This last bitstring (G_S) should result in generating a reversed order of default opcodes (G_1).

In this aspect the encoding has some similarity to structured genetic algorithms for function optimisation [163].

7.6.4 Main Optimiser Operations

This is the most important phase in the application of genetic algorithms because it involves three key steps: performing mutation and crossover operations, evaluating solutions via a fitness function and producing new generations.

7.6.4.1 Crossover Operation

The crossover operation is part of the generations reproduction process. This operation is carried out on pairs of genomes (parent) to combine parent's genes once the parents of a given generation were selected. A number of techniques have been in use to perform the operation, but the common technique is called the crossing over technique. The crossover technique can be applied using different approaches, such as single point, two point, cut and splice. Our algorithm is based on the one-point crossover approach which basically picks a random point on both parent genome bitstrings and then swaps the genes of the parents to produce new children as shown in Figure 7.4. The optimiser first applies one-point crossover technique on pairs of genomes "parent" from the selected $S - C$ genomes,

7.6.4.2 Mutation Operation

Mutation is a genetic operator that can be used to diversify genome representations from one generation to the next. This operation alters a few gene values (bits) in a genome from its original state. We use a bit string mutation technique which simply flips bits at random positions, and thus the probability of a mutation of m bits in a genome of length N is $\frac{m}{N}$. However, the number of bits (m) to be changed or the mutation probability should be set low to ensure that the search will not become a typical random search [128]. Our optimiser uses the flip bit mutation operator to invert 2 arbitrary bits from the new child genome.

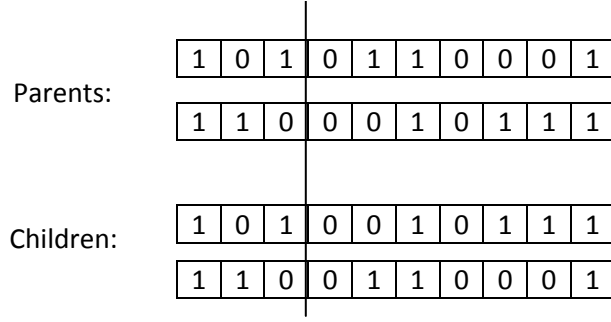


Figure 7.4: One-point Crossover

7.6.4.3 Permutation Operation

Using the presentations of genomes and permutation lists as described above in a regular sense give rise to problems with mutation and crossover operators because applying mutation and crossover on a given list is no longer necessarily a permutation. Our permutation machine design was based on a swapping technique which ensures that both binary crossover and mutation operations will lead to a valid permutation.

The permutation machine F reads in the bitstring G and an initial valid permutation L , carries out a sequence of valid permutations on L and outputs another valid Q permutation. That is,

$$Q = F(G, L)$$

Now if the permutation machine reads the identity permutation $I = 0, 1, 2, 3, \dots, N - 1$ then each permutation program G_j labels a permutation L_j produced by the application $L_j = F(G_j, I)$.

To illustrate how the permutation machine F works, let the permutation machine code be as follows:

$$G = b_{00}b_{10}b_{11}b_{20}b_{21}b_{22}b_{23}b_{30}b_{31}b_{32}b_{33}b_{34}b_{35}b_{36}b_{37}b_{40}\dots$$

The permutation machine F then proceeds according to the following rules: follows

- If b_{00} swap the 1st half of L with the 2nd half of L
- If b_{10} swap the 1st quarter of L with the 2nd quarter of L
- If b_{11} swap the 3rd quarter of L with the 4th quarter of L

- If b_{20} swap the 1st eighth of L with the 2nd eighth of L
- ... etc

Since the permutation of a permutation is still a permutation, and since swapping is a permutation operator, it is clear that the machine F will always produce a valid permutation of L if L is itself a valid permutation. Note that the first bit has more effect than the second and third, and that these have more effect than succeeding ones. In this aspect the encoding has some similarity to structured genetic algorithms [163].

It is also evident that the space that can be searched using this representation is of the order 2^2 which is less than $n!$ for all $n > 3$ so that the space searched by the GA based on this genome will only be a fraction of the possible permutation space. Since $n!$ is bounded above by $2^{n \log n}$ we could construct a permutation programme of length $n \log_2 n$ bits that would be capable of producing any permutation. How can we do this using our existing permutation function F ?

We need to apply $\log n$ independent permutations on the genome in sequence, and in order to do that we need a new function $g(i, s, p)$ which given an integer $i : 0..(\log_2 n) - 1$ a bitstring s as before and a permutation L will generate the permutation $\text{rot}(f(s, p), 2^i)$ that is to say it applies the permutation programme s to L as before and then cyclically rotates the permutation list by 2^i places. Suppose we have a genome of length $n \log n$ with $\log n$ bitstrings each of length n . We will denote the i^{th} of these component bit strings as s_i . The complete permutation space can then be scanned by composing g with itself $\log n$ times as follows

$$\text{for } i := 0 \text{ to } (\log_2 n) - 1 \text{ do } L = g(i, s_i, p)$$

However, for realistic numbers of opcodes (of the order of 100..200) 2^n already represents a huge optimisation search space and gives the GA plenty of scope to find improvements, so our initial experiments used a genome of length $2^{\lceil \log_2 n \rceil}$.

7.6.5 Generations Production

The process of generating a successive population (generation) j includes selecting fitter solutions from population $j - 1$ using a fitness function and then breeding new offspring using crossover and mutation operations on selected solutions.

7.6.5.1 Fitness Function

Typically, a fitness function is used to evaluate the efficiency of a given design solution. Our optimiser fitness function involves a complicated procedure and takes a considerable time to evaluate each solution. It is a multiple-step process. It involves building a code generator using individual generated solution one at a time and then compiling, running and measuring the performance of each training program. The following explanation describes these steps:

1. Produce a New Solution

Given a permutation list L , produce a new solution that is supposed to have a different instructions (opcodes) order and use the same order to define an instruction set for the target machine.

2. Rebuilt Compiler

Use individual solutions in a given population to reconstruct new code generators.

3. Compile Test Programs

If the code generator for that particular solution was successfully built, then compile the training programs.

4. Execute Test Programs

If a given training program was successfully compiled without any errors, then execute the training program.

The fitness function is designed to report any failure occurring during this process using different flags to distinguish each step. The flag is set to a negative value if any of the above steps failed. On the other hand, if the rebuilding, compilation and execution processes were completed without errors or any source of interruption, the function then

- Measures the execution time of every training program under each solution and reports the timings in files for further use.
- Gets the average execution time of all training programs under each solution in a population.
- Sorts the solutions in a given population in descending order based on the average execution time of the training programs.

7.6.5.2 New Generations

The production process of new generations starts from the point in which the fitness function stopped and goes through the following two steps:

- Selecting fitter solutions (Parents)

Given that the solutions in a population, say j , were already sorted in descending order, the optimiser will pick the fitter solutions or the parents of generation $j + 1$. The number of offspring is set by default to $\frac{1}{3}$ of the total number of solutions (a population), yet the optimiser allows users to determine the number of offspring to be breed. Using the default number of offspring, then the number of the fittest solutions to be selected as parents in a new generation ($J + 1$) is $\frac{2}{3}$ of the population j .

- Generating offspring (Children)

In this step, the optimiser will breed the remaining genomes. Breeding a child genome can be divided into several steps.

- The first step is to select a parent for the new born. Our optimiser is designed to choose the parents randomly given that each parent must have only one child; that is, a parent is used once and once only in the children reproduction process.
- The second step is to use a single-point crossover technique, as described above, to vary the genome of the new child. The crossover point is also selected randomly.
- The third and last step is to apply a mutation operator to preserve some genetic diversity between different generations. The optimiser inverts only 2 arbitrary bits from the new child genome.

7.7 Implementation

The optimiser was implemented in C++. The current version of the optimiser contains several C++ functions and uses a script program and a make file utility for building compiler code generators.

We start this section by introducing the key functions of the optimiser and then giving a brief introduction on the script program which was coded in AWK script language. We do

not intend here to present the entire code of these functions rather we give samples of the code that are essential in the discussion.

7.7.1 C++ Program

However, most of the optimiser implementation was written in C++. The C++ program responsibility starts from extracting the arguments (environment variables) that are passed from the command line, carrying out all genetic operations and ends up with documenting all the information used and obtained during the optimisation process.

7.7.1.1 Launching the Optimiser

To start the optimiser, users must provide the required information, especially the parameters that do not have default values such as the names of the training programs. The following command line shows how to call the program and what information is expected:

```
GA.exe -Tx -Gx -Sx -progTest1.pas -progTest2.pas -cpuPPC
```

where x must be an unsigned integer value. The symbols would be interpreted as the following:

- T: Prefixes a test trail's number which is used to create a directory as a work space for that test.
- G: Prefixes the number of populations and accordingly one subdirectory for each population will be created in the directory, which has just been created in the previous step.
- S: Specifies the number of solutions in a population and thus S files will be created to maintain all information on the different solutions in that population.
- prog: This symbol should prefix each training program name. No limit on the number of training programs that can be used in the fitness function.
- cpu: The last argument determines the target processor.

7.7.1.2 Key C++ functions

- Set Up:

It is the first function to be called. Its task is to extract data items passed via the command line, and based on these extracted parameters, it sets up a working place as shown in Figure

```
char newDir[50];
sprintf (newDir, "Test%d",T);
// Create a trail directory
if(opendir(newDir) == 0) {

    sprintf (newDir, "mkdir Test%d",T);
    system(newDir);

}
// Create a subdirectoy for each Generation
for (int i=0; i < NoPop ;i++){

    sprintf (newDir, "Test%d/GEN%d",T,i);
    if(opendir(newDir) == 0) {

        sprintf (newDir, "mkdir Test%d/GEN%d",T,i);
        system(newDir); }

    } else exit(1);
```

- Machine Instructions Opcodes

Its role is to get the identity permutation list of the target machine opcodes and count the number of opcodes.

- Create first population

The function is called only once to generate the initial population.

- Fitness function

The fitness function uses negative values as flags (*flg*) to signal failures of different processes. In the current implementation, for example,

-1 Indicates that the machine description file was not successfully modified.

-2 Hints at the failure of building the compiler code generator.

-3 Implies that the compilation of a give program failed

-4 Points out that the executions of a training program failed.

The following formulas determine the efficiency or fitness value of each design solution and the number of each activity or task that the fitness function does:

- The fitness of a solution is computed as

$$Fitnessvalue = \frac{C}{T}$$

where T is the average execution time of all training programs and C representing the completion success; i.e, $C = 1$ if $flg > 0$ otherwise $C = 0$.

- The number of reconstructed code generators

$$NCC = G + (G - 1) * (S - O)$$

where O represents the number of child in the generations $1..G$ while S , as defined above, represents the number of solutions and G the number of populations.

- The number of compilations that the optimiser is expected to do during the whole process is

$$NOC = NCC * noPrograms$$

where *noPrograms* is the number of training programs.

- New Generation production

This function triggers two other functions. First, it calls a function that chooses the best solutions as parents of a new generation providing that the solutions are already in descending order. It then calls a second function to generate offspring. The first function, however, chooses a parent randomly and then performs crossover and mutation operations to breed a new child and perform a slight change to the child genes.

- Rebuild a new code generator

This function goes over every design solution in each population, and for each solution it issues the following command line to invoke the AWK program for

.

7 Code Generator Optimiser

```
gawk -v T=TestNo -v G=Gen -v S=SolNo -f GA.awk
```

This command launches a program called `GA.awk` and passes the values `TestNo`, `Gen` and `SolNo` to the AWK variables `T`, `G` and `S` respectively.

- Documentation

The optimiser was also designed to report the status of almost every process and most of the obtained results in text files. The documented information includes:

- Generated genomes in every generation
- The exit status of modifying the machine specification files, building the code generators, compiling each training program.
- The execution time of each program under each individual solution.
- Average fitness value of each solution in a population.

7.7.2 AWK Program

The AWK language is a data extraction tool for processing files of text. The language looks to a file as a series of records that are split by newline characters and each record as a series of fields [164]. The `GA.awk` program is mainly designed to reconstruct the code generator of an individual solution, This information is used to determine the working space. The AWK program performs the following tasks:

- Read machine opcodes from a file and then group them in a list
- Modify the machine description file and report the process status.
- Invoke a makefile for building the compiler code generator
- Report if the compiler building process succeeded or failed.

7.8 Experimental Results

For comparing the efficiency of the optimiser on different machines, we run it on the PowerPC code generator, which is a sub-set generator for the Cell Broadband Engine, and on the IA32 architecture with SSE2 instructions. The VP front end compiler, however, was the same in both architectures and the performance of each design solution was tested using the same benchmarks on both machines.

7.8.1 Benchmarks

We tested the optimiser using two sets of benchmarks: one set contains three benchmarks used in the fitness function to evaluate the algorithms, and the second set consists of two benchmarks used to test whether the optimised design solution, which the optimiser produced, yields the same performance improvement on these two programs. The benchmarks of the fitness function are: n-body, prime sieve and a validation program that was developed for testing a range of VP array operations.

- N-Body Problem

The n-body problem is a simulation of particles moving under the influence of gravity. The program starts with an initial position, mass and velocity of a group of particles at a given time. It then uses that data to work out the motions of all particles and to calculate their positions at later times. The calculation is based on the laws of motion and gravitation. The program was taken from the Programming Languages Shootout web-site (<http://shootout.alioth.debian.org/>).

- Sieve Program

The sieve program finds all prime numbers that are less than or equal to a given integer value “ n ” using Eratosthenes’ method. The algorithm works by first creating a list of the integer numbers > 1 and $\leq n$, and then continuously removing composite numbers until eliminating all composite values and ending up with only prime numbers.

- Vector Operations

The last application is a special purpose program that was developed to test the Vector Pascal compilers on various array operations, such as transpose, reduction, dot product operations, of different data types such as single precision floating-point, single precision fixed-point and characters.

We selected the benchmarks that use different data types. In the n-body program, most operations are performed on single precision float-point values while the prime sieve program mainly uses single precision fixed-point values. The validation program, however, was designed to test VP array operations on various data types. This mixture of data types, which the benchmarks use, provides some sort of uniformity in using the average fitness value as a means of evaluating the performance of a design solution.

The other applications, which were used to test the optimised design solution, are Mandelbrot and Spectral-norm. They were also picked up from the Shootout benchmarks. The

first application is a mathematical set of points which usually requires a large number of computations, and it is commonly used to evaluate runtime performance. The second application is a program that calculates an eigenvalue using the power method.

7.8.2 Environment Variables

Here are the main parameters used in these experiments:

- The optimiser was set to run for 6 generations; i.e.,—*G*6.
- The size of the working population was set to 90; i.e.,—*S*90.
- The Vector Pascal training programs were Sieve, n-body and vecTest, and they were set as follows: `-progSieve.pas` `-prognBody.pas` `-progVectest.pas`
- The number of offspring was not specified, and therefore the optimiser will use $\frac{1}{3}$ of the solutions *S* as new children.
- The optimiser was designed to carry out 2 bits mutation. Since the number of instructions in both machines range between 180-250, the mutation probability then would be around 0.01.

In these experiments, we run the optimiser for only 6 generations for two reasons. First, the fitness function sometimes took a long time as it had to go through several steps to do each test. Also, the time to complete the whole process varies from one design solution to another because the algorithm sometimes either converges on an inappropriate solution or has difficulty converging at all. The second reason is that it did not seem that we get any considerable improvement after the 4th generations as we shall see shortly in the diagrams shown in the following section.

7.8.3 Results

The following results show the average and the best fitness values of each population on both machines. The fitness value is proportional to the performance of the training programs, and it is computed as $\frac{c}{t}$ where $c = 1$ if the programs all compiled and 0 otherwise and t is the average run time of the programs. The higher the fitness scores the shorter the average execution time of the tested applications. The results also include a table to show the genetic algorithm improvements of several applications on the PowerPC machine.

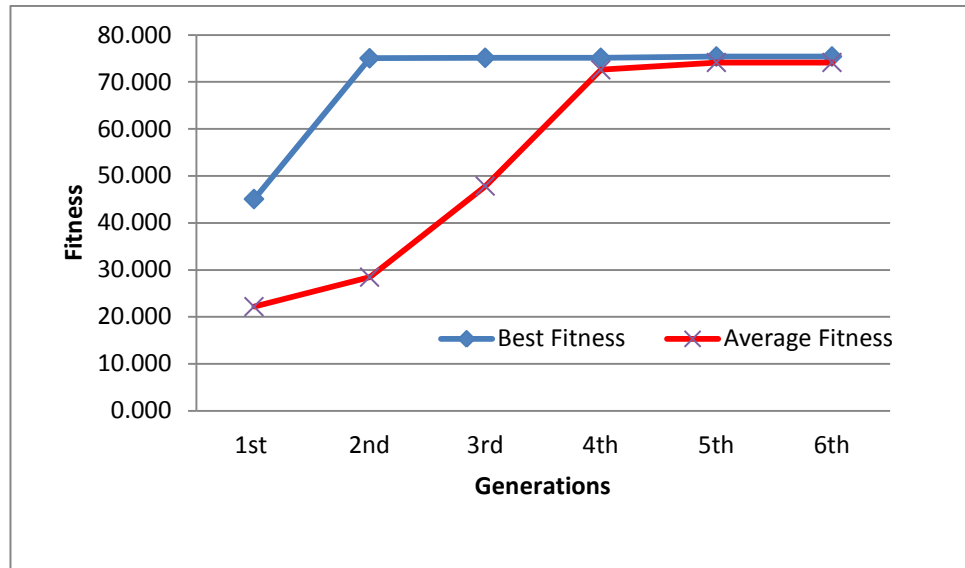


Figure 7.5: The performance of a genetic algorithm on optimizing a PowerPC instruction set ordering.

- PowerPC Code Generator

We performed the experiments on the master (PowerPC) processor of the Cell Broadband Engine (Cell BE) found in the Playstation 3. The PowerPC machine description includes 184 instructions. The code generators are generated using different solutions (IS ordering) offered by different generations. Figure 7.5 shows the average fitness and the best fitness of the solution in several populations. The average fitness values reflect the performances of the code generators on three different applications. The higher the fitness scores the shorter the average execution time of the selected applications. This figure generally reveals that the performances of the PowerPC code generators improved significantly throughout the first three successive generations. The results also demonstrate that the solutions offered by the last generation lead to performance improvement by a factor of 3.4 as compared to the solutions provided by the first generation. The average fitness value of the solutions of the first generation was about 22.2 while it reached around 72.6 at the fourth generation with an improvement factor of about 3.2. However, in the last two generations, the algorithm became steadier and an insignificant improvement was gained during these two generations.

Figure 7.5 also plotted the best solution in each generation. The chart depicted that the best solutions were produced in the first two generations, and that the second generation provided the best design solution among all generations.

The results presented so far showed that the optimised ordering provided a con-

```
li r6, -24
lwax r5,r6,r4
```

Figure 7.6: Register-to-Register Operation

siderable improvement on the training programs, but the question is can that be generalised to applications that were not in the training programs set?

Tables 7.1 shows the results of our investigation of whether the improvements gained as a result of using the optimizer best-produced solution on the PowerPC machine can be generalised to applications other than the training programs. The table shows the genetic algorithm improvements on applications other than the training set. It compares the performances of five applications using a default instruction ordering and the best order (solution) that was generated by the optimiser. The first three applications in the Table are the training programs, and the last two applications; Mandelbrot and Spetual-Norm, are arbitrary applications that were used to see if the best solution can lead to the same improvements gained on the training programs. The results shown in Table 7.1 reveal that improvement on non_training programs is considerably close to those on the training programs. Yet, the enhancement on the Mandelbrot application is not as tight to those achieved on the training programs as the Spectual-Norm is.

The possible reasons for performance degradation on the Mandelbrot could be associated with loop and arrays addressing optimisations. The first reason in this regard is that loop executions often involve boundary checking and addressing, and these two operations usually require using offsets. The second point is that PowerPC machines offer only 16-bit immediate arithmetic and logical instructions which are normally used in offset-based operations. The alternative on the PowerPC architectures, however, for 16-bit offset operations is to use register-to-register instructions instead of using register-immediate instructions. Figures 7.6 and 7.7 show a simple example for loading a 32-bit word, say a loop or an array lower bound, into the register (r5) from an address relative (-24) to register (r4). The assembly code in both figures does the same task. Yet the code in Figure 7.6 includes double the instructions shown in Figure 7.7. This simple example shows that the register-to-register approach require more instructions than the other approach, and consequently it will most likely be more costly. Thus using inefficient instruction ordering may result in generating code that gives correct results but inadequate in terms of performance improvement.

```
lwa r5, -24(r4)
```

Figure 7.7: Register-to-Immediate Operation

Set	Program	Performance in (Sec.)		
		Default	Optimised	Improvement
		Order	Order	
Training Programs	Sieve	26.5	20.6	22%
	N-Body	39.2	30.5	22%
	Vector Operations	30.6	23.2	24%
Test Programs	Spectral-Norm	83.2	63.2	24%
	Mandelbrot	24.7	21.1	15%

Table 7.1: Genetic Algorithm Improvements on the PowerPC. The first set of applications were training programs and the last two programs were not in the training set.

The third reason in regard to the degradation performance on the Mandelbrot is that in these experiments the Mandelbrot program was executed only a few times while the other applications each was executed 1000's of times. And therefore, the Mandelbrot program does not involve as many loop iterations as the other applications, and therefore, it did not benefit from the optimisation of loop executions and array addressing as much as the other applications.

- Pentium Code Generator

The Pentium machine description includes more instructions than the PowerPC. It consists of 252 instructions. The Pentium's code generator, unlike the PowerPC's code generators, had been under development for some years and had received a considerable manual enhancement. Thus, the augmentation in the performance due to the optimiser is not expected to be as good as on the PowerPC. Figure 7.8 shows that the average fitness of the first three generations was increased slightly. It then declined on the forth generation, but it was still better than the first generation. After that, the average fitness started improving but with very small margin. The diagram also shows that the best solution was not improved during the six generations. This could be due to the Pentium instruction set being well tuned.

Figure 7.9 depicted a normalised performance gained by optimizing instruction orderings of both machines. The chart shows that the reordering of the PowerPC's instruction set improved the performance of the PowerPC around 3 times while on Pentium the optimizer reports a slight improvement.

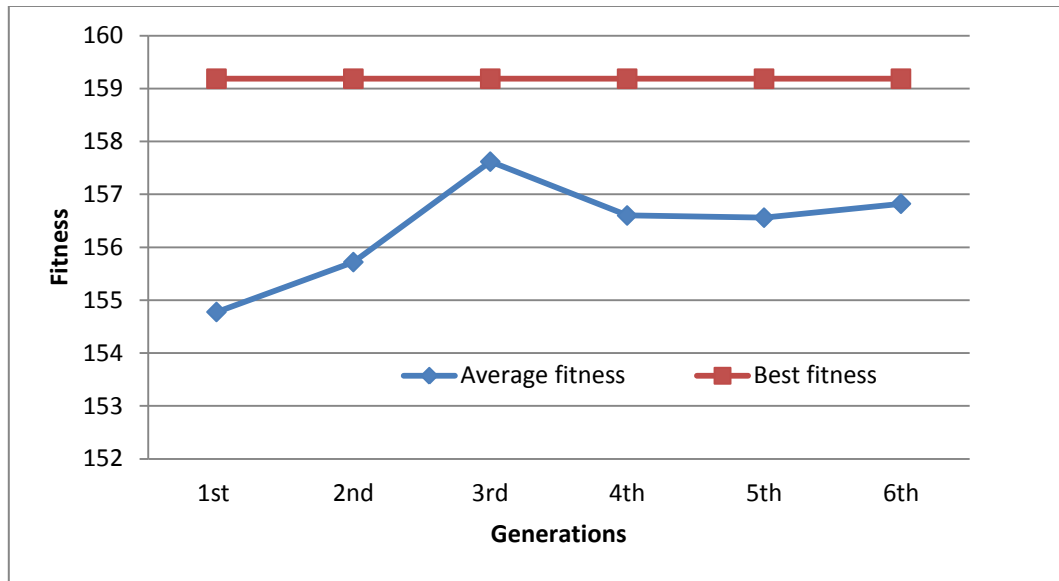


Figure 7.8: The results gained by optimizing previously manually optimised code generator for the Pentium using a genetic algorithm. The average fitness values reflect the performances of the code generators on the three selected applications.

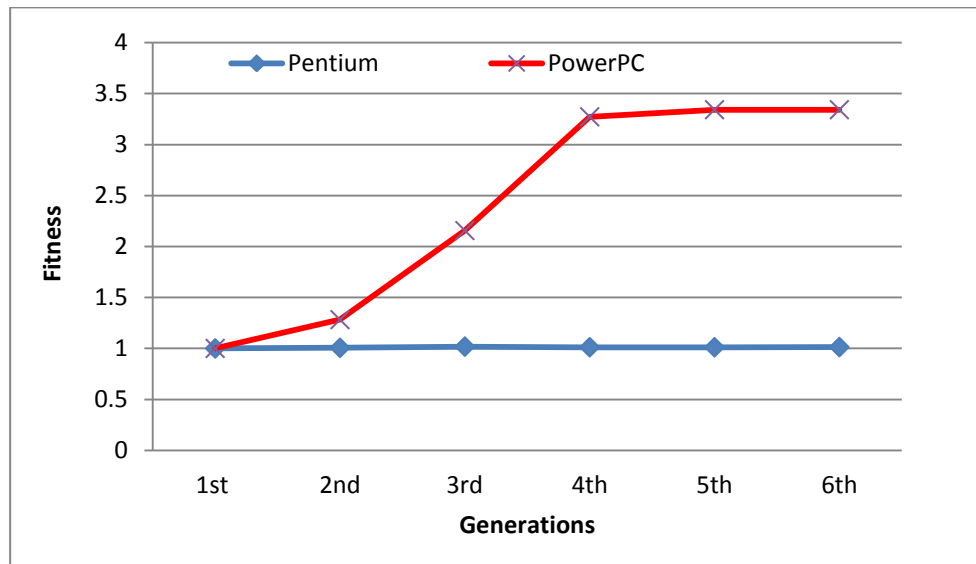


Figure 7.9: Normalized performance gained by optimizing instruction set ordering of PowerPC and Pentium 4 architecture using a genetic algorithm.

7.8.4 Conclusion

- During the experiments, it was noticed that the algorithm sometimes either converges on an inappropriate solution or has difficulty converging at all. This situation normally occurs during the first few generations, but the number of cases, which do not converge, decreases as the algorithm during building new generations excludes the weakest solutions. This can be clearly seen in Figure 7.5 which showed the optimizer functioned well on the PowerPC machines because it was not well tuned.
- One conclusion is that the proposed approach showed that GA can assist compiler designers in tuning and enhancing code generators, and the results also show that scheduling instruction sets can improve the performance of compilers.
- The current implementation of the optimiser may not terminate if the execution of a training program does not terminate, and therefore, it will be a good idea to add a time boundary to terminate processes that take more time than expected, such as compiling or running a training program.
- The current version of the optimiser design allows users to run the optimiser several times (trials) on the same machine. The optimiser also documents most of the steps that it performs, such as generated genomes, corresponding solutions and the results of the fitness function, in separate files for each trial.
- Finding an optimised solution in large search space is usually expected to take considerable time to finish, especially if it runs for many generations, or if it uses many training programs to evaluate the fitness of each solution. This problem can be handled by this optimiser, for example, if the optimisation process is interrupted for any reason, users can easily resume the process from any point (population) given that the preceding population is already assessed. This feature basically copies all required information from the last population to the following population. After that, the optimiser can resume the selection, reproduction and the evaluation processes on the last generation and continue with the consecutive generations.

8 Evaluating VP-Cell Compiler

In this chapter, we start with outlining the configurations of the machine used in these experiments, describing in brief the tests that were conducted to validate the developed VSM and the extended compiler and then depicting the main measures that were taken to improve the performance of these two parallelising tools.

After that, we present the experimental results obtained using our developed parallelising tools, the VSM model and the VP-Cell compiler, and discuss the achieved performance and scalability. We run two sets of experiments, micro-benchmarks and real world applications, to examine if our VP-Cell compiler improves the execution time of the automatically parallelised programs as compared to the execution time of the sequential ones. We also run a number of C micro-benchmarks to examine if the VSM model can be used as an independent tool to parallelise C code and if it provides any performance improvement.

8.1 Machine Configuration

We used a Sony PlayStation 3 (PS3) console for the experiments. The details of its configuration are summarised in Table 8.1.

Console Mode		Play Station 3
Number of Core		9 cores
Master Core (PPE)	Number	1
	Memory	256 MB
	Speed	3.2 GHz
	Cache	256 KB L2, 64 KB L1
Accelerators (SPEs)	Number	6 available
	Memory	256 KB
	Speed	3.2 GHz
Operating System	Linux	Fedora 7
IBM Cell SDK		3.1.0
GNU Chain Tool		4.1.1

Table 8.1: Machine Configuration

The VSM interpreters are compiled using GNU ppu-g++ and spu-g++. The SPE code is based on the second version of the main SPE management library; libspe2, which basically provides a flexible threading model and collaborates with the pthread library.

8.2 Testing Developed Tools

During the course of this project, there were numerous tests run on the original PowerPC compiler, the VSM mode and the extended compiler. These tests include:

- Testing the PowerPC Compiler (Sequential Version)
 - Writing sometimes simple C code to check the structure of the generated assembly code. This was needed with the lack of documentations on the GAS assembler.
 - Writing small C programs to see how parameters are passed. This is critical issue in functions calling conventions to permit separately compiled routines in the same or different languages to interact with each other.
 - Writing Vector Pascal programs to look at and trace ILCG intermediate code and assembly outputs. This is sometimes needed to see how intermediate generated code looks specially when a compiler does not converge or there is a segmentation fault.
 - Using the gdb debugger to analyse or trace generated code when problems were encountered.
 - Testing the PowerPC back-end compiler using three different test suits:
 - * Standard routines that include more than 1300 lines of code written in Pascal. The compiler should pass this test before it goes to the next tests.
 - * Pascal Validation Suite version 5.7 which includes more than 190 programs which was introduced by the British Standards Institution in 1982.
 - * Vector Pascal Acceptance test that contains more than 50 additional testing programs.
 - Changing the machine description if a given program failed to compile for the PowerPC but compiled successfully by previously developed VP compilers for other architectures.

- Testing the Instruction Ordering Optimiser of the PowerPC and Pentium architectures. A paper on the optimiser is due to be submitted in Feb. 2012.

Based on the last version of the compiler, out of 190 tests in the Pascal Validation Suite, 83% passed the validation tests, 11% passed but outputted a FAIL, 2% passed but go in infinite loop when executed and 4% failed to compile. The compiler was also tested using 50 VP Acceptance tests, and only 8% of these tests were failed.

- Testing the VSM Model
 - In order to identify the VSM performance problems and obstacles, we run many tests which time the activities of each instruction individually to be able to break down time spent in each activity.
 - Writing a simple C simulator to test the latency of different functions.
 - Conducting many experiments to determine the appropriate VSM register size,
 - Testing the two techniques for sending messages through mailboxes. We gain a considerable improvement by using MMIO register instead of IBM SDK library functions.
- Testing the Extended Compiler
 - Testing the order of the extended machine instruction set.
 - Using the DDT graphical debugging tool to debug application on the Cell Processor. DDT was developed by Allinea for multithreaded and parallel applications. It was a trial version, which was extended for free, yet it was very helpful.

8.3 Performance Tuning

It was necessary to consider optimising the two main components of our compiler system to improve the performance of the individual modules. This work involved:

- Tuning the PowerPC Compiler

The large bottleneck in the development of VP back end compilers is the instruction ordering. The order of machine instructions can affect the performance of the generated code, and as expected compilers have to go through this process many

times specially during the early stages of their development. Thus, scheduling machine instructions manually is uninteresting work and represents the bottleneck to improve the performance of our developed compiler because the search space was enormous. At the beginning of compiler development, the reordering of the PowerPC instructions was done manually, but in the later stages we depended on the Instruction Ordering optimiser which we developed as part of the project. The optimiser offers around 20% improvement, as we shall see in Chapter 7, on the tested applications as compared to our hand-tuned machine instructions.

- Tuning the VSM Model
 - Optimising SPE Code: Was able to reduce the SPE program size by using the same code to load and store data of any data type.
 - Virtual Register Size: Tested various register sizes, 1024, 2048, 4096, 8192 and 61384, to find the optimal virtual register size.
 - Messaging Procedure: Managed to reduce the latency time to send two 32-bit messages by a factor of around 13x.
 - Optimising Store operation: The order of DMA transfers involved in Store operation was set in a way that allows overlap between data transfers.
 - VSM Vectorization: Using SPE intrinsic function provides substantial improvement on computational operations.

8.4 Experimental Results

The results, which are presented in this section includes, were obtained using

1. A set of micro-benchmarks includes Basic Linear Algebra Subprograms (BLAS). The micro-benchmarks were implemented in Vector Pascal to evaluate the VP-Cell compiler system's performance and scalability.
2. The same set of micro-benchmarks but were coded in C. This experiments demonstrate and examine the possibility of using VSM as an API by other languages.
3. Two real world applications.
 - a) The first application looks at the computational efficiency of the Cell processor on the N-body problem. This problem is a scientific simulation that involves computing the motion of a number of planets (bodies) under physical forces such as gravity.

```

Line 1: PROGRAM GeneticTest;
Line 2: type vec=array[1..4096] of real;
Line 3: var i:integer; s:real; x,y,z:vec;
Line 4: begin
Line 5:     x:=1.23; y:=2.34; z:=0.0;
Line 6:     z:=x+y;
Line 7: end.

```

Figure 8.1: Simple Program in VP to Test a BLAS Kernel

- b) In the next example, we shall look at an image filtering application. Convolution filters are important tools to process images for certain features such as smooth or sharpen an image.

In these experiments, we assume that the arrays in the source programs are not subject to any memory alignment restrictions and their lengths are integer multiple of the VSM register size. All the benchmarks run on single-precision floating point arrays using VSM registers of size $4096 * P$ bytes where P is the number of SPEs unless otherwise stated.

8.4.1 VP Micro-Benchmarks

The following experiments were the first Vector Pascal benchmarks to be compiled by the VP-Cell compiler system. The selected micro-benchmarks include classical vector operations such as reduction operation and dot products of two vectors and typical examples of BLAS-2 such as a matrix-vector product and rank-1 update operations.

Figure 8.1 shows the VP program that was used to evaluate the BLAS1 and BLAS2 kernels. It is a very simple program that includes variables declarations and initializations and an array expression which adds vectors x and y and saves result in vector z . It is very important to note here that the same code was used for generating sequential code and parallel code without any changes, annotations or directives.

8.4.1.1 Selected Kernels

Table 8.2 lists the examined BLAS kernels and operations involved in each kernel. All the variables in this table are single-precision float point variables in which s scalar, x, y and z are vectors of size n and A is a square matrix of size $n \times n$. The last column in the table indicates whether the kernel involves a blocking operation or not. This table

Description	Expression	Operations Involved					Kernel Mode
		load	store	Mul.	Add	Sqrt	
Replicate a scalar	$x := s$	$\sqrt{^*}$	$\sqrt{}$	x	x	x	NB
Square root of a vector	$x := \text{sqrt}(y)$	$\sqrt{}$	$\sqrt{}$	x	x	$\sqrt{}$	NB
Vector reduction (+)	$s := \text{redp}(x)$	$\sqrt{}$	$\sqrt{^*}$	x	$\sqrt{}$	x	B
Elem-to-Elem product	$x := y * z$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	x	x	NB
Dot product of 2 vectors	$s := y \cdot z$	$\sqrt{}$	$\sqrt{^*}$	$\sqrt{}$	$\sqrt{}$	x	B
Matrix-Vector Product	$y := A * x$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	x	B
rank-1 update	$A := A + x * y^T$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	x	B

*Scalar Operation , B: Blocking Mode, NB: Non blocking Mode

Table 8.2: Basic Linear Algebra Kernels

shall help to explain and discuss the performance results attained by the VP-Cell compiler, but before discussing these results we would like to explain how the compiler offloads or evaluates an array expression or a kernel on the SPEs.

8.4.1.2 Parallelising Array Expressions

During the compilation process, our compiler splits every expression in the source code that comprises arrays of sizes equal or bigger than the VSM registers into a sequence of VSM instructions (or operations). Every VSM instruction is then transformed into three assembly instructions. The first two generated assembly instructions pass the required information to the PPE while the third instruction calls the proper PPE routine. Upon executing the code, if one SPE is used, the called PPE function then sends all the data to that SPE; otherwise it chops the data into blocks by calculating the starting address of each block, and then sends a message to each SPE. The message(s) must determine the required operation, the virtual register(s) to be used and the starting address in case of memory access. For example, to evaluate the expression; $x := y + z$, on the SPEs, the PPE splits the expression into four operations: load vector y , load vector z , add vectors y and z , and finally store the sum in x . The code segment given in Figure 8.2 shows an example of VP-Cell compiler-generated machine code that corresponds to array expression $x := y + z$.

8.4.1.3 Results

In the micro-benchmarks, we used floating point arrays of size 4096 and virtual SIMD registers of size 4KB and run each micro-benchmark 10^5 times just to get fair measures.

Figure 8.3 shows the performance of the PPE versus one SPE on the BLAS-1 and BLAS-2 micro benchmarks, which are shown in Table 8.2. The achieved speedups on the different

```

// Machine registers r22,r23 and r24 hold the starting
// addresses of vectors x, y and z respectively.
// load vector y into virtual register 0

    li 3, 0
    la 4,0(r23)
    bl LoadVec

// load vector z into virtual register 1

    li 3, 1
    la 4,0(r24)
    bl LoadVec

// Add virtual registers 0&1 and keep result in 0

    li 3, 0
    li 4, 1
    bl AddVec

// Store the returned value into x (address r22)

    li 3, 0
    la 4,0(r22)
    bl StoreVec

```

Figure 8.2: Generated Code for Expression $x := y + z$

kernels using one SPE compared to the PPE range from 4 times to 9.3 times. It is worth, however, mentioning here that the current version of the VP compiler for the PPE does not generate vector instructions.

Figure 8.3 reveals that significant speedups was achieved using a single SPE; the SPE was more than 9 times faster than the PPE on the Replicate kernel and 8 times faster on Element-to-element multiplication kernel. The SPE attained its highest speedups on these two kernels because they do not involve, as shown in the last column in Table 8.2, a call to a blocking operation. However, the highest speed on the Replicate kernel is mainly due to the size of data to be transferred in each kernel. On the Replicate kernel, the SPE was required to copy only a scalar from the main memory while on the Element-to-element multiplication kernel, the SPE was required to copy two vectors.

This figure also shows that the SPE performed adequately well on the Square Root, Reduction and Rank-1 Update kernels with an average speedup of around 6x, yet it was not as good as its performance on the first two kernels. The cause of this performance degradation differed from one kernel to another. If we start with the Square Root operation, the SPE speed was slow because this operation is considered as a complex SPE operation [165] which takes more time in the computation than arithmetic operations. The SPE was

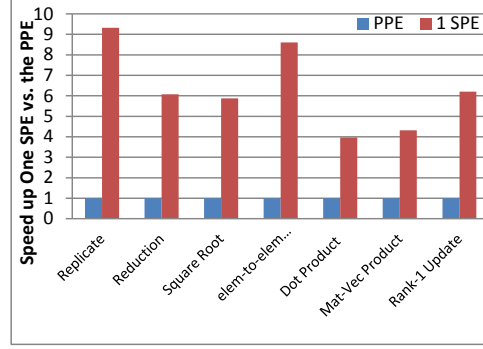


Figure 8.3: (a) Performance PPE vs. SPE

also slow on the Reduction and Rank-1 Update kernels but the reason is not associated with operational complexity as in the Square Root operation. The Reduction and Rank-1 Update kernels involve a blocking operation; namely the reduction operation. The reduction operation is relatively costly compared to the other operations because the PPE has to wait until all the SPE complete the operation and then collect and sum the results which were returned by the SPEs. It is also very important to report that despite the Rank-1 Update kernel involves more operations than the reduction kernel, the SPE speedup on both kernel was very close. This is due to the fact that the PPE performed poorly on the Rank-1 Update kernel, and this shows that the SPE performance was relatively fast. The SPE performance was, however, the poorest on the Dot product and Matrix-vector kernels because both involve a reduction operation and require more data movements than needed in the other kernels.

To explore how the VP-Cell compiler parallelises array expressions automatically on multiple SPEs, we run the same kernels in parallel on 2 and 4 SPEs. We only used 2 and 4 due to a divisibility constraint on the length of the virtual register. In these experiments the virtual register size was 4KB, and therefore this size is not integerly divisible by either 3 or 6. Also we could not use 8 SPEs because there are only 6 SPEs accessible on the PS3.

Figure 8.4 (a) shows a significant improvement in the performance can be achieved by using multiple SPEs. This figure plots the speedups obtained from using 1, 2 and 4 SPEs compared with the PPE performance on each kernel. The figure shows that the performance when 2 SPE were used was about 16 times faster than the PPE on the Replicate kernel and about 29 times with 4 SPEs, and almost the same speedup was achieved also on the element-to-element multiplication. Though the speedup on the other kernels ranges between 12x to 21x, the average speedups achieved on 2 SPEs was about 11.5x and around 20x one 4SPEs.

8.4 (b) presents the same results shown in Figure (a) but from different perspective. This figure shows the scalability attained by using multiple SPEs. As we can see, the SPEs

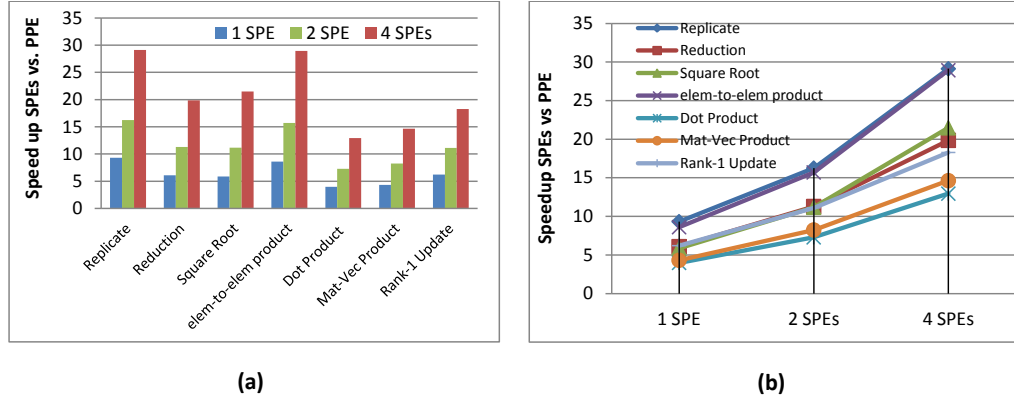


Figure 8.4: Performance and Scalability of SPEs

achieved near-linear scalability on all the kernels with an overall speedup of factor $P^{0.84}$ where P is the number of processors. The maximum speed up gained on the Replicate kernel was by a factor of $P^{0.94}$ and the minimum was about $P^{0.75}$ on the dot product of two vectors.

8.4.1.4 Conclusion

The results on the VP micro benchmarks shown in Figure 8.3 and in Figure 8.4 are noteworthy; considering that the SPE has a small memory and requires data to be transferred using DMAs. Moreover, the array expression in these micro-benchmarks involve a limited number of arithmetic floating point operations, and each expression has to store the result back to the PPE. Thus the store operation represents between 20% to 33% of the operations involved in each kernel such as the “sqrt” operation. This operation requires Load, a computational operation; namely square root, and store. Thus, the store operation accounts for $\frac{1}{3}$ of the involved operations and therefore dominates the whole process as it is relatively more costly than load and computation. However, we expect the compiler to attain better performance on array expressions in which the percentage of involved blocking operations as compared to the total operation is small. That is, we expect an expression has several operands and one store.

8.4.2 C Micro-benchmarks

The following C micro-benchmarks and the VP version given in the previous subsection are exactly the same. The C version includes classical vector operations such as reduction operation and dot products of two vectors and typical examples of BLAS-2 such as a matrix-vector product operation. I run these selected C kernels on the PPE as well as the


```

// Sequential Matrix-Vector Product function
void seqMatrixVecProduct() {
    for (int r=0;r<size;r++)
        for (int c=0;c<size;c++)
            v3[r]+=M[r][c]*v1[c];
}
// Parallelise Matrix-Vector Product on SPE using the VSM
// routine "dotpf" to parallelise the operation.
void parMatrixVecProduct() {
    for (int r=0;r<size;r++)
        v3[r]=dotpf(M[r],v1,size);
}

```

Figure 8.5: C code to compute the dot product of two vectors in Sequential on the PPE and in parallel using the Cell's SPEs.

SPEs using VSM as an API to investigate if the C parallelised code results in any performance improvement and then evaluate the performance scalability of the SPEs. All the kernels run using single-precision float point variables in which S scalar, $v1$, $v2$ and $v3$ are vectors of size 4096, M is a square matrix of size 1024×4096 , the SPE virtual registers size was 4KB, and most of micro-benchmarks run 10^5 times just to get fair measures.

8.4.2.1 Parallelising Array Operations

Figure 8.5 illustrates how a C BLAS kernel can be parallelised using the current VSM implementation. This figure shows the implementations of the C functions which were used to evaluate in sequential and parallel a BLAS2 kernel, in particular the product of a matrix and a vector. The first function in Figure 8.5 is a simple sequential C function which uses two nested `for` loops to iterate over the rows and columns in order to compute the dot product of each row in matrix M and vector $v1$ and store the results in vector $v3$. The second function, called `parMatrixVecProduct()`, uses the VSM model as an API to parallelise and evaluate operations on the SPEs. It iterates over the rows of matrix M , and in each iteration it invokes a PPE routine, called "dotpf". The PPE routine in turn dispatches the received request to the SPEs to compute the dot product of two vectors and returns a scalar. It is very important to note that the code does not include any annotations or directives, and in order to use the VSM model programmers are only required to call the appropriate PPE function.

We compiled these two functions using the same GNU compilers which were used to compile and build the VSM object files. The sequential C code was fully optimised using the -O3 mode and run on the Cell's master (PPE) processor. To parallelise the second function, the VSM object file should be linked to the C code using a single command line as sh

```
ppu32-g++ foo.cpp ppe.o spe.o -lspe2 -lm -ofoo.exe
```

8.4.2.2 Results

Figure 8.6 plots the performance achieved from running the C kernels on the PPE and the SPEs. The C kernels corresponds to the VP micro benchmarks shown in Table 8.2. The performance of one SPE on most of the kernels, as shown in Figure 8.6, was poorer than the PPE but not on the Replicate and Mat-Vec product kernels. The one SPE was faster on these two kernels than the PPE by a factor of 1.5x. The improvement on the Replicate kernel was due to two reasons: first this kernel does not require a great deal of data movement because it only involves sending a scalar instead of moving an entire vector to an SPE local memory. Secondly, the SPE carries out the Replication operation using SIMD instructions via the SPE intrinsic functions which offer a considerable speedup. The improvement on the Mat-Vec product kernel, however, is due to first the slightly slow performance of the PPE on this kernel relative to other kernels. Secondly, the SPE is only required to return a scalar which needs only one 128byte DMA transfer instead of as least three DMA transfers which are needed to store back an entire vector as was explained in Section 5.6.4.1.

Consider now using 2SPEs, when the C code was parallelised via the VSM model on 2 SPEs, the performance of the SPEs on all the kernels was in general better than the PPE performance. The speedup obtained using 2SPEs on the dot product kernel was around only 0.05% higher than the PPE, but it was much better on the other kernels. For example, on the Replicate operation the 2SPEs' speedup was about 2.4x while on the Mat-Vec Product kernel reached 2.7x. The average speedup on the other kernels was approximately 1.3x.

Moving to 4SPEs, as one can see the SPEs performance achieved using C code is much better than the PPE. The 4SPEs achieved a speedup of more than 3.3x and 3.9x on the Replicate and Mat-Vec product kernels respectively. On other kernels, the average speedup was increased from 1.3x when 2SPEs were used to 1.8x with 4SPEs.

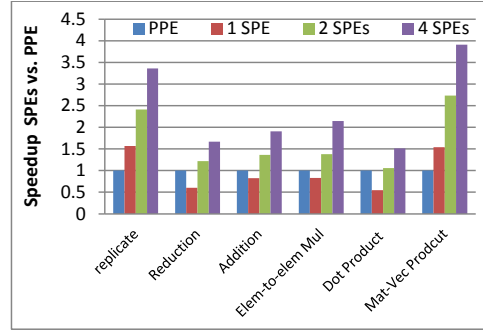


Figure 8.6: Performance of C code on the PPE and the SPEs by Using VSM as an API

Figure 8.7 plots the scalability achieved by running the selected C BLAS kernels as the number of SPEs increases. The SPEs, as shown in this figure, achieved near-linear performance on all the kernels. Notice that the two kernels on which the one SPE performed poorly were linearly scaled when 2SPEs were used. The interpretation of the linear improvement on the Reduction kernel is that this kernel returns only a scalar value instead of an entire vector, and thus the computation dominates the whole process rather than memory access and communication overheads. And once the computation was carried out on 2SPEs in parallel, the computation time was reduced; that is, an optimal tradeoff between communication and computation was achieved.

Figure 8.7 also shows that a full linear speedup gained on the Reduction kernel using 2SPEs. The speedup factor was almost 2 times faster than the one SPE, and it was the maximum speedup achieved among the other kernels. Yet, the performance on the same kernel was not linearly scaled as we moved from 2SPE to 4SPEs, and this was due to the increase of the communication overheads associated with sending messages and gathering results from 4 sources instead of 2SPEs. The speedup factor attained running the Reduction kernel on 4SPE instead of 2SPEs was about only 1.4 times. The same reasoning accounts for the improvement gained using the dot product kernel basically because it also does not need to restore a whole vector.

However, an overall speedup factor of more than 1.7x obtained when moving from one SPE to 2SPEs and by a factor of approximately 1.4x as we moved from 2SPEs to 4SPEs.

8.4.2.3 Conclusion

The current implementation of the two vectors dot product kernel can be improved further. The implementation as shown in Figure 8.5 is a naive implementation because in every iteration it will load vector v_1 . This implementation could be optimized by introducing a new separate PPE routine which takes a matrix and a vector. The new routine can then

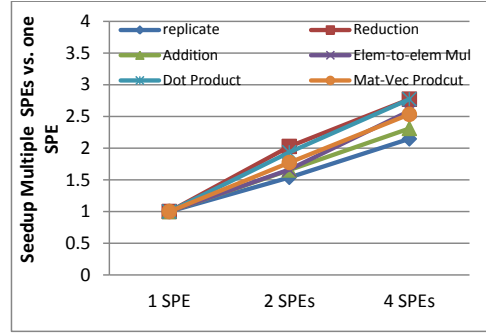


Figure 8.7: SPE Scalability On BLAS Kernels Using C Code.

handle this deficiency internally by loading the vector only once on the SPEs by placing the load vector operation outside the loop.

In this experimental code, the C kernels was parallelised by simply calling PPE routines, as shown in Figure 8.5 and in Figure 8.2 to examine the performance of the parallelised code. Figure 8.7 shows marked improvements can be achieved by parallelising C code on multiple SPEs using the VSM model considering that VSM was not implemented to be used as an API. Note also that the C code was parallelised without adding any annotations or directives or handling thread creation, data partitioning and synchronization.

The experimental results in this section and the previous subsection also demonstrated that a full implicit parallelising tool, such as the VSM model, can be used as intermediate layer for developing parallelising compilers or can be used as API for parallelising array operations that are written in any programming language that can be compiled separately and linked to C code.

8.4.3 N-body Benchmark

The N-body problem is a scientific simulation that involves computing the motion of a number of planets (bodies) under physical forces such as gravity. The gravitational force between each pair of bodies is defined by their position, velocity and mass. N-body simulations usually require massive computing power, and hence a number of experiments have used this problem for evaluating machine performance. The N-body problem has been also used recently for comparing the performance of modern parallel technology such as the new SIMD extension supported by the Sandy Bridge processor [143] and other parallel architectures such as GPUs [166]. This problem was also selected by the Scottish Informatics and Computer Science Alliance (SICSA) research body as a challenging problem in Phase II of the SICSA Multicore Challenge [167]. Figure 8.8 presents the main steps of this problem.

8 Evaluating VP-Cell Compiler

```
For  $N$  bodies
Each time step

    For each body  $B$  in  $N$ 
        Compute force on it from each other body
        From these derive partial acceleration
        Sum the partial accelerations
        Compute new velocity of  $B$ 
    For each body  $B$  in  $N$ 
        Compute new position
```

Figure 8.8: N-body Problem Pseudocode

8.4.3.1 N-Body Algorithms

The N-body simulation is required to compute the force between each pair of bodies. In reality, the number N of bodies or particles is often very large, and thus a number of algorithms and methods have been developed to optimise the simulation. The two common algorithms for computing the total force on each body are the All-Pairs method and Barnes-Hut Treecode [166]. The total number of interactions needed to be computed using an ordinary approach, such as All-Pairs algorithm, is N^2 while Barnes-Hutt method is an $O(N \log N)$ algorithm [166].

This benchmark drawn from the Great Computer Language Shootout which was originally contributed by Christoph Bauer [168]. In the Vector Pascal version I made explicit use of operations on whole arrays, but to express the problem in parallel style using arrays, the full N^2 forces have to be computed. The VP parallel algorithm is similar to the All-Pair method approach because it also goes over each body in N , say B , and computes the forces of all other bodies $N - 1$ on the body B . This additional computation associated with the parallel solution ensures that the calculations are independent and can be safely carried out on multiple processors in parallel. The main part core of the algorithm is a procedure, called advance, whose main loop uses a position matrix to compute a matrix of acceleration components for each body in N . These components are summed along the rows to yield a final velocity increment. The algorithm 8.1 was used to test the VP-Cell compiler on the Cell processor.

8.4.3.2 PPE vs. SPE Performance

In order to use the VSM model, which uses large vector registers, we had to scale the original code from 5 bodies to 1024, 4096, 8192 and 16384 bodies and create C and Vector Pascal versions to compare between the two implementations. In the Vector Pascal

Algorithm 8.1 N-Body Parallel Solution Procedure

The types used are:

```

vect =array [1..n ] of realt ;
pvect = ^ vect ;
matr =array [1..3,1..N ] of realt ;
pmatr = ^ matr ;

```

The variables used are:

```

Let x ∈ pmatr; x,y,z coordinates of system
Let v ∈ pmatr; x,y,z components of velocity
Let a ∈ pmatr; x,y, z components of acceleration
Let di ∈ pmatr; x,y,z components of distance from body i
Let sqdi ∈ pvect; vector of sum of square distances from i
Let d ∈ pvect; vector of distances from i

```

The Key procedure:

```

procedure advance ( dt : real );
var

    Let i, j ∈ integer;
    Let t ∈ real;
    row: array [1..3] of real ;

begin

    for i ← 1 to N do row ← x↑[t0, i];
    // Compute the displacement vector between each planet and planet i.
    di↑ ← x↑ - rowT ;
    // Next compute the euclidean distances
    sqdi↑ ← di↑[1] × di↑[1] + di↑[2] × di↑[2] + di↑[3] × di↑[3];
    // Prevent divide by zero for the square distance from self
    sqdi↑[i] ← 1;
    d↑ ← √sqdi↑ ;

    //Now compute the acceleration vectors
    a↑ ← m↑ × di↑ / (sqdi↑ × d↑) ;
    for j ← 1 to 3 do
        v↑[j, i] ← v↑[j, i] + dt × Σ a↑[j] ;
    // Finally update positions.
    x↑ ← x↑ + v↑ × dt ;

end ;

```

version, we made explicit use of operations on whole arrays as shown in the previous algorithm, and we run the same VP code on the Cell processor in sequential and parallel. In the following discussion, we shall use the Kilo unit, such as 1K, 2K, 8K and 16K, to refer to the N-body problem size. Note that if the data size to be processed on an SPE is bigger than the SPE virtual register, then the compiler will automatically unroll the operation and use multiple DMA transfers.

Figure 8.9 shows the speedups obtained using the PPE and a single SPE on different sizes of the problem. Comparing the performance of the two core types, we can see that the speedup factor gained from using one SPE compared to the PPE was around 3.6x when the problem sizes were 1K, 4K and 8K bodies and 4.5x with 16K bodies. To analyse these results let us start with the 1K problem. This problem simply needs 4KB and since we are using only one SPE, the SPE managed to use the optimal register size, which is 4KB, and this is why we see the 1SPE performed better on 1K than on 4K and 8K. However, on the 4K and 8K problems, the SPE needed multiple DMA transfers to process the whole data which consequently introduces additional overheads. This explains the slight drop in the speedups we see in Figure 8.9 on the 4K and 8K problems. Moving to the 16K problem, though the 16K bodies requires double the DMA transfers used to process the preceding size, there was an increase in the speedup factor relative to the three smaller sizes. This was, in fact, due to the poor performance of the PPE as shown in Table 8.3, and caching could be the main factor which accounts for the PPE shortfall.

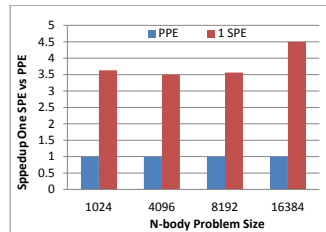


Figure 8.9: Performance of One SPE vs. the PPE (N-body Problem)

8.4.3.3 SPE Scalability

Before we discuss the results from using multiple SPEs, it is worth to mention here that the maximum VSM register size that can be used for the N-body problem of size 1024 bodies is 4KB, and therefore when multiple cores are used the data is partitioned equally on the SPEs registers. On the other sizes of the problem, however, the SPE virtual registers size is 4KB. Recall that if the data is bigger than an SPE virtual register, then multiple DMA transfers will be used. We shall see how the number of DMA transfers and the register sizes may affect the performance of the SPEs.

Table 8.10 presents the speedups of the SPEs compared to the PPE on the four selected sizes of the N-body problem. The table shows that the SPEs performance on the 8K problem was irregular yet on the other sizes of the problem the performance proceeds improving in two dimensions: vertically as the number of SPEs increases and horizontally as the size of problem gets bigger.

N-body Problem Size	SpeedUps		
	1 SPE/PPE	2SPEs/PPE	4SPEs/PPE
1024	3.6	5.9	7.9
4096	3.5	6.2	10.3
8192	3.6	6.1	9.9
16384	4.5	7.6	12.4

Figure 8.10: Speedups of Vector Pascal on the Cell's processors (N-body Problem)

To depict and have a good picture on how the SPEs performance is scaling we present the same measurements given in Table 8.10 graphically in Figure 8.11 (a) and (b). Figure 8.11 (a) presents the performance of the SPEs compared to the PPE on each size of the N-body problem. We start with the 1K problem which requires arrays of 4KB. Due to the size of the data in this problem, the maximum VSM register size that one can use is 4KB, and therefore the data size to be processed on multiple SPEs gets smaller as more and more SPEs are used. For example, with 4SPEs and 1K bodies, each SPE ends up processing only 256 bodies or 1KB of data, and thus we should not expect the same performance improvement as the SPEs are increased. Actually, this can be simply perceived in Table 8.10 by comparing the improvement gained on the 1K problem as we move from 1SPE to 2SPEs and then to 4SPE. The performance on the 1K problem was improved by a factor of 1.6x as we moved from 1SPE to 2 SPEs but by only 1.3x as we moved from 2SPEs to 4 SPEs. This means that the computation of small data blocks does not pay off the communication and DMA overheads, and that is why multiple SPEs, as we explained in the previous subsection, did not perform on 1K bodies as good as on the other three sizes of the problem due to the data size.

Now consider the 4K and 8K problems which were solved using arrays of size 16KB and 32KB respectively. Figures 8.11 (a) show that the performance on both 4K and 8K sets is generally very close. The SPEs performance on the 4K is yet very slightly better than that of 8K because only half the DMA transfers used compared with 8K and consequently half the DMA overheads. This can be seen very clearly by comparing the performances of the 2SPEs and 4SPEs under the 4K and 8K problems; see the results in the second and

8 Evaluating VP-Cell Compiler

the third rows in Table 8.10. These results indicate that the performance degradation rate, which was affected by doubling the number of DMA transfers each time, was less than 2% when 2SPEs were used and about only 4% with 4SPEs.

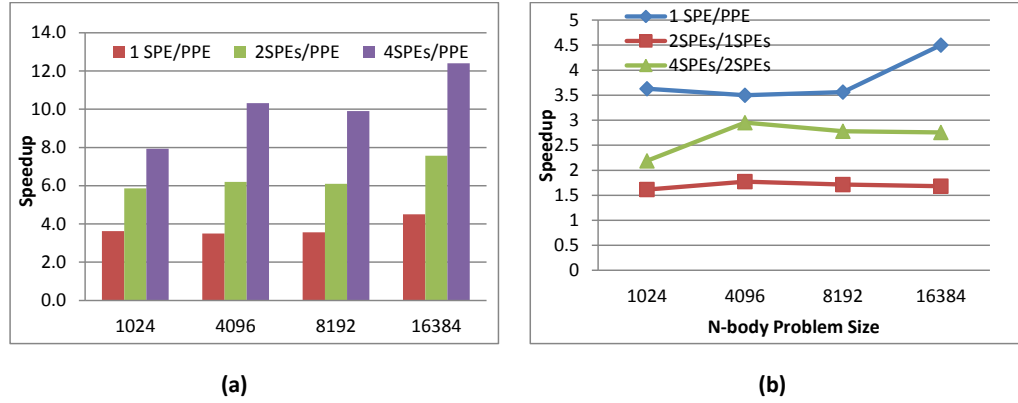
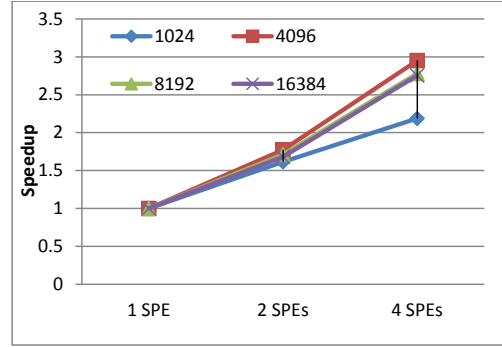


Figure 8.11: The SPEs Performance

However, The 4K problem compared with 8K and 16K should introduce the best environment for scaling the performance on multiple SPEs for two reasons. First the size of the problem is large enough to allow up to 4 SPEs to use the optimal register size, which is 4KB, and the second reason is that the 4K problem requires less DMA transfers than of the 8K and 16K. Yet, Figure 8.11 (a) shows that the SPEs fastest performance was achieved on the 16K problem rather than the 4K problem even though the SPEs used 4 times as many DMA transfers as with the 4K problem. The performance improvement reported on the 16K problem was mainly due to the poor performance of the PPE on 16K compared to its performance on the smaller ones.

Figure 8.11 (b) shows how the SPEs behave as the size of the problem increases. It shows the speedups gained from using 1SPE as compare to the PPE performance and also the performance of 2SPEs and 4 SPEs relative to 1SPE. Figure 8.11 (b) shows that on the first three sizes of the problem the performance obtained from using 1SPE compared to the PPE decreases as the size of the problem gets bigger, and this is normal due to the increase in the number of DMA transfers, but on the 16K problem the SPE behaved differently. The speedup obtained from using 1SPE compared to the PPE on the 16K problem rapidly jumped from 3.6x to 4.5x where it should have been declined because it requires double the DMA transfers that of 8K. This shows clearly the effect of the PPE's poor performance as explained in the previous subsection. The chart in figure (b) also shows that the 2SPEs and 4SPEs performed as they were expected. Their performance started dropping as the size of the problem gets bigger. The drop of speedup reflects the cost of the additional DMA transfers as we move from one size to the next. However, 8.11 (b) reveals that the



(a)

Figure 8.12: SPE Scalability

2SPEs and 4SPEs achieved the best performance on the 4K problem as they were using the optimal register size.

Figure 8.12 shows that the scalability attained from using multiple SPEs is grown in a near-linear fashion. This figure reveals that the climax speedup obtained using 4 SPEs on the 4K N-body problem was by a factor of around 2.95x, and an overall speedup of factor $P^{0.80}$ obtained on the 4K problem where P is the number of processors. On the contrary, the minimum speedup was achieved on the 1K problem, and this was due to the small size of the data blocks. The average speedup factor on the 1K problem was about $P^{0.63}$.

8.4.3.4 Conclusion

Thus these results indeed demonstrate why 4K register on each SPE would be the best size to achieve better performance. This assumption, however, is based on the current VSM implementation, but the optimal size could be reduced if the VSM model can be optimised further. We can also conclude here that the sizes of VSM register and the SPE virtual register have more impact on the performance improvement compared with the increase in the number of DMA transfers.

8.4.3.5 VP vs. C Performance

The following comparison is presented just to show the competence of two different high-level programming languages in exploring parallelism automatically in sequential source code and exploiting the performance potential of the Cell processor. We compare the performance of the Cell on the N-body problem using Vector Pascal and C code. The code in both languages was written in a sequential style. The C code was compiled using the

same GNU compilers which were used to compile and build the VSM model. Since we only use sequential code, the C code runs only on the PPE under the optimisation modes; -O0 and -O3. While the same VP code runs on the Cell master processor and the SPEs. To parallelise the VP code, users are only required to set the target processor, such as PPC or SPE, as a command line variable.

Just to note here that the VP code, which is based on the All-Pairs algorithm to solve the N-body problem[166], requires N^2 operations while the C code is based on the Barnes-Hutt method which uses only $O(N \log N)$ algorithm [166]. The two programs were run for 20 iterations. The results shown in Table 8.3 represent the performance per iteration in seconds using the two languages and on four different sizes of the problem. We used exactly the same VP results that we have just discussed in the previous subsection. Recall also that on the 1K problem (4KB of data) the SPE virtual registers size is only 1KB, but on the sizes other than the 1K, each SPE uses the optimal register size, which is 4KB.

Table 8.3 shows that VP performed almost the same as the unoptimised C code (-O0) on 1SPE. Comparing VP with unoptimised C, the VP performance was actually running 20% slower on 1K bodies, but as the size of the problem gets bigger the VP performance started improving and getting very close to C. On the 16K problem, however, VP performed better by using 1SPE than the unoptimised C. Yet, if we compare the performance of VP on 1SPE and C with full optimisation (-O3), VP could not compete with optimised C code. Though VP slowness continues through all the sizes, the SPE deficiency factor dropped more than 40% by going down from 2.3x to only 1.3% as we move from 1K to 16K. However, VP, on 2SPEs, performed slightly better than the unoptimised C code on most sizes of the problem and performed almost the same compared to the C optimised version.

Now let us consider the performance of VP and C on 4SPEs. Table 8.3 shows that the VP-Cell compiler performed much better than the C compiler when 4 SPEs were used to solve N-body problem of size 4K or bigger. Figure 8.13 provides graphical representation on the performance of VP versus C using 4 SPEs. VP was slightly slower than fully optimized C, and this is, as we mentioned previously, due to that each SPE works on a small data block. On the other sizes of the problem, VP performance much better than C, and the speedup factor obtained by running VP on 4SPEs ranges between 1.6 and 2 times faster than the C code.

8.4.4 Images Filtering

The convolution is an image filtering operation that can be applied to smooth or sharpen an image based on the neighboring pixels. The neighbor pixels are given some values as

N-body Problem Size	Performance (sec per Iteration)					
	Vector Pascal				C	
	PPE	1 SPE	2 SPEs	4 SPEs	O3	O0
1024	0.381	0.105	0.065	0.048	0.045	0.085
4096	4.852	1.387	0.782	0.471	0.771	1.328
8192	20.355	5.715	3.334	2.056	3.232	5.591
16384	100.250	22.278	13.248	8.086	16.524	25.86

Table 8.3: Performance of Vector Pascal vs. C.

Performance of Vector Pascal vs. C (N-body problem). The code in both languages is written in a sequential style. The VP code uses N^2 operation to solve the problem while C used only $O(N \log N)$ operation.

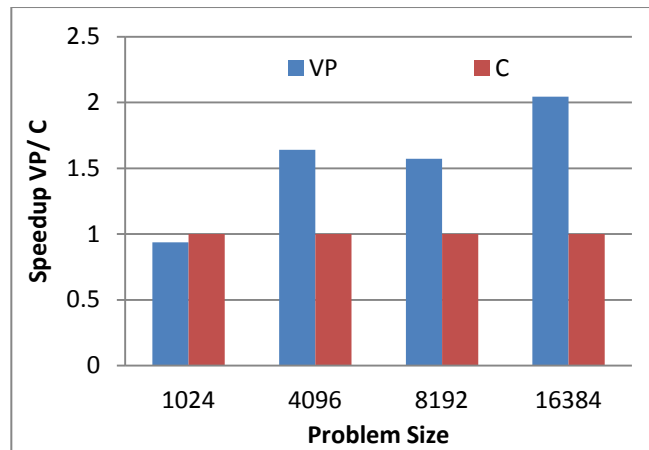


Figure 8.13: Vector Pascal and C Performance

Vector Pascal and C Performance on the Cell processor using 1KB SPE virtual registers to solve the 1K problem and 4KB SPE virtual registers for 4K, 8K and 16K.

weight to each pixel, and these values are usually presented as a matrix called convolution kernel. This experiment demonstrates the implementation of a parallel convolution algorithm on a matrix of real numbers. The format of the program that follows is generated by the built in literate programming tool of the compiler which outputs listings in formatted L^AT_EX.

The following procedure, *fblurtime*, is the main body of the program. It simply specifies the size and type of data that will be operated on and then times the Vector Pascal convolution procedure; *blurp*. The arrow (\uparrow) here refers to pointers.

```

program fblurtime ;
const

    size =1024;
    runs =30;

type

    pmat =  $\wedge$  matrix ;

var

    Let im  $\in$  pmat;
    Let t1, t2  $\in$  double;

procedure pconvp ( var p:matrix; c0,c1,c2:real ); (see Section 8.4.4.1 )
procedure blurp ( var im:matrix ); (see Section 8.4.4.2 )
procedure cconv ( var p:real ; rows,cols:integer ; c1,c2,c3:real );
external ;
begin

    new ( im,size,size );
    t1  $\leftarrow$  secs;
    for i  $\leftarrow$  1 to runs do

        blurp (im $\uparrow$ );

    t2  $\leftarrow$  secs;
    writeln( 'PASCAL' , t2 - t1);

end .

```

8.4.4.1 pconvp

Convolution of an image by a matrix of real numbers can be used to smooth or sharpen an image, depending on the matrix used. If A is an input image, K a convolution matrix, then if B is the convolved image

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j}$$

A separable convolution kernel is a vector of real numbers that can be applied independently to the rows and columns of an image to provide filtering. It is a specialisation of the more general convolution matrix, but is algorithmically more efficient to implement. If \mathbf{k} is a convolution vector, then the corresponding matrix K is such that $K_{i,j} = \mathbf{k}_i \mathbf{k}_j$.

Given a starting image A as a two dimensional array of real values, and a three element kernel c_1, c_2, c_3 , the algorithm first forms a temporary array T whose elements are the weighted sum of adjacent rows $T_{y,x} = c_1 A_{y-1,x} + c_2 A_{y,x} + c_3 A_{y+1,x}$. Then in a second phase it sets the original image to be the weighted sum of the columns of the temporary array: $A_{y,x} = c_1 T_{y,x-1} + c_2 T_{y,x} + c_3 T_{y,x+1}$.

Clearly the outer edges of the image are a special case, since the convolution is defined over the neighbours of the pixel, and the pixels along the boundaries are missing one neighbour. A number of solutions are available for this, but for simplicity we will perform only vertical convolutions on the left and right edges and horizontal convolutions on the top and bottom lines of the image.

```

procedure pconvp ( var p :matrix ;c0 ,c1 ,c2 : real );
var

    Let  $T, l \in \text{pmat}$ ;
    Let  $i \in \text{integer}$ ;
    Let  $r, c \in \text{integer}$ ;

begin
    // This sequence performs a vertical convolution of the rows of the plane p and places the result in the temporary plane T.

     $r \leftarrow p.\text{rows}$ ;
     $c \leftarrow p.\text{cols}$ ;

    // Allocate a temporary array initialised in the middle to the original one.

    new (  $T, r + 2, c + 2$ );
     $T \uparrow[2..r + 1, 2..c + 1] \leftarrow p_{1..r, 1..c}$ ;

    // Replicated out the rows to fill the top & bottom margins of array T.

     $T \uparrow[1] \leftarrow T \uparrow[2]$ ;
     $T \uparrow[r + 2] \leftarrow T \uparrow[r] + 1$ ;

```

Now perform a vertical convolution of the plane T and place the result in p . Note that this is done by multiplying the whole temporary array by the kernel constants and then adding shifted versions of it.

```

 $p_{1..r, 1..c} \leftarrow T \uparrow[2..r + 1, 2..c + 1] \times c1 +$ 
 $T \uparrow[1..r, 2..c + 1] \times c0 +$ 
 $T \uparrow[3..r + 2, 1..c + 1] \times c2$ ;

 $T \uparrow[1] \leftarrow T \uparrow[2]$ ;
 $T \uparrow[r + 2] \leftarrow T \uparrow[r] + 1$ ;

```

again place it into the temporary array and this time replicate horizontally

```

 $T \uparrow[2..r + 1, 2..c + 1] \leftarrow p_{1..r, 1..c}$ ;
 $T \uparrow[] [1] \leftarrow T \uparrow[] [2]$ ;
 $T \uparrow[] [c + 2] \leftarrow T \uparrow[] [c + 1]$ ;

```

Next perform a horizontal convolution and free the temporary buffer.

```

 $p_{1..r,1..c} \leftarrow T \uparrow [2..r + 1, 2..c + 1] \times c1 +$ 
 $T \uparrow [2..r + 1, 1..c] \times c0 +$ 
 $T \uparrow [2..r + 1, 2..c + 2] \times c2;$ 

dispose (  $T$  );
end ;

```

8.4.4.2 blurp

This procedure just specifies that the parallel blurring uses the kernel [0.25,0.5,0.25].

```

procedure blurp ( var im : matrix );
begin
  pconvp ( im ,0.25,0.5,0.25);
end ;

```

8.4.4.3 Results

To investigate the effectiveness of the VSM register sizes, the following discussion first analyses the performance of the compiler on the different sizes of the images using a 4KB VSM register on one or multiple SPEs, and then explores the compiler's performance using optimal register sizes. The VSM register size was chosen to be 4KB to show the effects of operating on small SPE virtual registers also goes with all the sizes of the images. Note here that by setting the VSM register to 4KB, the size of SPE virtual registers will depend on the number of used SPEs. Thus, the SPE virtual register size will be 4KB when one SPE is used, 2KB if 2SPEs are used and 1KB with 4SPEs. We run this experiment using three different sizes of images that consists of 1024x1024, 2048x2048 and 4096x4096 square pixels (floating point) format, and the results, which are presented in the following table and figures show that significant performance improvements are obtained by using the Cell's accelerators on the different sizes of VSM registers.

The performance of the Cell's cores on the blurring of different image sizes using only a 4KB VSM register is shown in Table 8.4. All the measurements were taken using the Linux command "time" and the unit of measurements is seconds.

Image Size		Performance (sec) - 4K VSM Register			
		PPE	1SPE	2SPEs	4SPEs
1024	1024	0.20	0.11	0.08	0.09
2048	2048	1.95	0.41	0.27	0.32
4096	4096	8.86	1.56	1.04	1.24

Table 8.4: Performance of the PPE and SPEs Using Different VSM Register Sizes

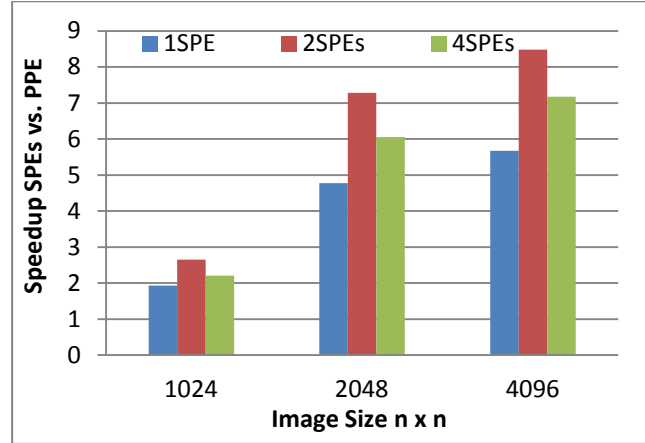


Figure 8.14: Performance of SPEs versus PPE on Blurring Program

Figure 8.14 compares the speedup obtained using the Cell's SPEs against the PPE performance as the number of the SPEs increase. Consider first the performance of a single SPE compared to the PPE. As we can see in this figure and also in Table 8.4, the PPE started performing poorly as the size of the image gets bigger, and we have seen similar behaviour in the previous example when the size of the N-body problem was 16K. The performance of one SPE using 4KB register, which is the optimal size, was about 2 times faster than the PPE. However, as the size of the image gets bigger and bigger, the SPE's speedup relative to the PPE performance looks as if it started increasing more rapidly. The SPE speedup jumped from about 2x on 1024 image to around 5x and 8.5x on 2048 and 4096 image size respectively. This high jump in the speedup, however, was due the slow performance of the PPE on big size images. Actually, this can be simply deduced from Table 8.4 as the PPE was 10 times slower on 2048 image than on 1024 while the SPE on the same sizes was around only 4 times slower. The figure also shows that a single SPE and 2SPEs interestingly performed better than the 4SPEs, and we have also seen a similar situation with the N-body benchmark when the performance on the 1K problem was degraded as we moved from 2SPEs to 4 SPEs.

Scalability is investigated under different configurations. Figure 8.15 shows the perfor-

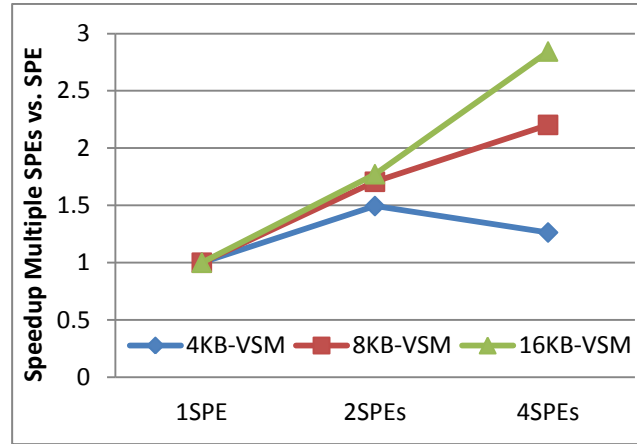


Figure 8.15: SPEs Scalability on Blurring 4096x4096 Image Using Different VSM Register Sizes

mance of blurring an image of size 4096x4096 pixels as it is scaled across the Cell's SPEs using 4KB, 8KB and 16KB VSM registers. This figure, which shows the result of splitting the image on 1, 2 and 4 SPEs, reveals that when the SPEs use an optimal register size, this leads to better scaling of the application. For example, with 4SPEs and using 4KB VSM register (or 1KB register per SPE), the performance of the 4SPEs compared to 1SPE was dropped from 1.5x to 1.3x. Yet, the performance of 2SPEs was the best when 8KB VSM register (or 4KB register per SPE) was used, while the best performance of the 4SPEs was reached when 16KB VSM register (or 4KB register per SPE).

8.4.5 Conclusion

We have presented here the results obtained from running two sets of benchmarks which include 12 BLAS1 and BLAS2 kernels and two real world examples on the Cell processor to compare the execution time of sequential and parallel code written in Vector Pascal and C programming languages. The VP-Cell parallelising compiler targets only array expressions, and thus the Vector Pascal code must be array based implementation. Actually, this is why we choose only two real applications because only few real world benchmarks coded in Vector Pascal, and many of the Pascal benchmarks are coded in a standard sequential fashion but not array-based code.

These experimental results show that significant performance improvements can be obtained by using the Cell's accelerators especially if the SPEs use the optimal register size. The results also show that the VSM model in its current implementation is a very beneficial tool for parallelising intensive-data applications.

9 Conclusion and Future Work

A number of modern computer systems have been offering high performance platforms as with reasonable cost that made them widely used in many application areas, such as image processing, graphics, multimedia, modeling and scientific computation. However, this widespread industry adoption of multi-core, homogeneous and heterogeneous, architectures has a significant influence on mainstream software and applications development and has introduced new challenges for software developers to provide proper, simply-used and up to date tools for developing parallel and concurrent programs. The emergence of the new heterogeneous platforms nowadays makes it even harder for compilers to generate efficient parallel code than with homogeneous machines. Heterogeneous multi-core architectures, such as the Cell heterogeneous architecture, have different types of processing cores, and each type is designed to support and carry out a different set of functions.

Our research project aimed at developing a new approach for automatic parallelisation of computations on large data sets that is specifically targeted towards modern heterogeneous hardware. The developed tool focused only on array-based code and should be capable to parallelise vector/array operations to run on general purpose heterogeneous multicore platforms. The ambitions were to reduce the complexity associated with the fully automatic parallelisation approach by focusing only on array expressions, and also to ease the task of developing programming parallel applications by concentrating on algorithms rather than on parallelisation issues such as communication, partitioning, alignment, and synchronisation.

The work was demonstrated by designing and implementing a Virtual SIMD Machine (VSM) model that hides the intricate details of the Cell heterogeneous architecture completely. This model was based on new parallelisation techniques that imitate a SIMD instruction set using virtual (large) registers and Virtual SIMD Instructions (VSIs). VSM supports parallelising array operations across the heterogeneous cores of the Cell processor. It consists of two co-operating virtual interpreters, one for each of the two core types, PPE and SPEs, on the Cell processor. The PPE interpreter runs on the Cell's master core type and manages the system overall and offers a set of functions for parallelising array operation on the SPEs as well as other operations such as thread creation and message handling.

Throughout this thesis we also investigated whether an array programming compiler, such as the Glasgow Vector Pascal (VP) compiler, can be extended to use this model to exploit data parallelism implicitly and attain sufficient performance. The work involved extending the machine description of the PowerPC back end compiler to be capable of collaborating with our VSM model. The compiler extension included defining a virtual SIMD register set and introducing a new instruction set that operates on these virtual registers. We also had to modify some machine-dependent routines such as ENTER and LEAVE to create and terminate threads. This arrangement simplifies the task of compiler code generators to transform sequential code into parallel code.

This dissertation has shown that imitated SIMD techniques can be used to develop a fully implicit parallel programming model which reduces the burden of developing parallelising compilers that exploit a heterogeneous multi-core architecture. The work in research project presented a tool that does not require learning a new language, using produced hints to make an efficient use of processing units such as in CellVM or even annotation or directives such as in OpenMP, OffloadC++ and OpenCL. The overall results in this thesis demonstrate this approach significantly reduced the execution time of the automatically parallelised programs compared to the execution time of the sequential ones without the need for any annotations or process directives. Our approach also showed considerable performance improvements related to scalability. However, the VSM performance is below the theoretical peak performance, which is mostly due to communication overhead and alignment constraints as the SPEs are designed to operate on 128-byte aligned data. The VSM design also imposes divisibility restrictions on the size of SPEs virtual registers and the number of SPEs that can be used, and this results in not being able to use all the resources (SPEs) of the Cell processor.

9.1 Contribution

- Demonstrating the possibility to abstract and completely hide the intricate details of heterogeneous architectures.
- Representing techniques that provide a fully implicit programming model for parallelising array operations and support scalable parallelization.
- Introducing a framework that can be as an abstract model to shorten the time for developing parallelising compilers.
- Developing parallelizing tools that aid compilers to automatically generate parallel code with the aim of reducing the execution time of the parallelized code.

- Developing a new approach to optimize the code-generator of a compiler. This approach is based on genetic algorithm techniques to automatically optimise machine instructions ordering.

9.2 Future Work

There are a number of interesting future directions that may enhance the VSM model. Some of these opportunities for future work are listed.

- While the current implementation of the VSM targeted at the Cell BE processor, the concept is more general and can be applied to heterogeneous multicore systems with a host with accelerators (APUs) such as AMD Fusion, Intel Ivy Bridge, and GPUs.
- The communication overhead is still high. Thus using optimised the communication overhead would be one way to improve the VSM performance and reduce the virtual register sizes; i.e, the size of arrays which could be adequate for parallelisation.
- The VSM instruction does not back dual-mode operations such as multiply-accumulate operations, and thus supporting this feature shall evidently improve many computations which involve accumulation operations such as vector dot product and matrix multiplications operations.
- The VSM current implementation includes an instruction for every computational operation for every data type. Each computational instruction uses two physical machine registers and two assembly instructions in order to pass the virtual register numbers to the analogous PPE function, and the PPE then gives a code to operation (opcode). However, it would be more efficient if a VSM instruction uses only one register to pass the virtual register numbers as well as the opcode. This results in first reducing the number of assembly instructions by $\frac{1}{3}$ and most importantly to implement only one PPE (template) function for all data types.
- The code size of the current SPE interpreter (or program) is about 25KB which is considerably good since it requires only 10% of the SPE local memory. However, template techniques would be useful also to optimise the SPE interpreter code size, especially if more operations are added.

Bibliography

- [1] S. Gill, “Parallel Programming,” *The Computer Journal*, vol. 1, no. 1, pp. 2–10, 1958.
- [2] J. Glasgow and M. JenkinsCarl, “Expressing parallel algorithms in Nial,” *Parallel Computing*, vol. 11, no. 3, pp. 331–347, 1989.
- [3] C. Geoffrey, R. Williams, and P. Messina, *Parallel Computing Works*. San Francisco, CA: Morgan Kaufmann Inc., 1994.
- [4] W. GregoryV, “The History of the Development of Parallel Computing,” 1994.
- [5] A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel virtual machine: a users’ guide and tutorial for networked parallel computing*. MIT Press Cambridge, MA, USA, 1995.
- [6] K. Pingali, “Parallel programming languages,” tech. rep., Cornell University, 1998.
- [7] P. Cockshott and K. Renfrew, *SIMD programming for Windows and Linux*. Springer, 2004.
- [8] A. Scheinine, “Introduction to Parallel Programming Concepts,” tech. rep., Louisiana State University, 2009.
- [9] K. Asanovic and et la., “A view of the parallel computing landscape,” *Commun. ACM*, 52(10), pp. 56–67, 2009.
- [10] P. Mckenney, M. Gupta, M. Michael, P. Howard, J. Triplett, and Walpole, “Is Parallel Programming Hard, and if so, why?,” tech. rep., Portland State University, 2009.
- [11] G. Tournavitis, Z. Wang, B. Franke, and B. M., “Towards a Holistic Approach to Auto-Parallelization,” in *ACM, PLDI09*, October 2009.
- [12] R. Leupers, “Code selection for media processors with simd instructions,” in *DATE ’00: Proceedings of the conference on Design, automation and test i Europe*, (New York, NY), pp. 4–8, ACM Press, 2000.

- [13] F. Luebke and G. Humphreys, “How gpus work,” *IEEE Computing*, pp. 126–130, 2007.
- [14] M. Hassaballah, S. Omran, and Y. Mahdy, “A review of SIMD multimedia extensions and their usage in scientific and engineering applications,” *Comput. J.* 51, pp. 630–649, 2008.
- [15] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate gpu vs. cpu performance without the answer,” International Symposium on Performance Analysis of Systems and Software (ISPASS), 2011.
- [16] A. Krall and S. Lelait, “Compilation techniques for multimedia processors,” *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 347–361, 2000.
- [17] N. Srereman and G. Govindarjan, “A vectorising compiler for multimedia extensions,” vol. 28, pp. 363–400, 2000.
- [18] A. Richards, “The Codeplay Sieve C++ Parallel Programming System,” 2006.
- [19] A. Ewing, H. Richardson, A. Simpson, and R. Kulkarni, *Writing Data Parallel Programs with High Performance Fortran*. Edinburgh Parallel Computing Centre, 1998.
- [20] “Parallel programming essentials via the intel tbb,” <http://www.codeproject.com/KB/Parallel Programming/Threading Blocks.aspx>.
- [21] R. G. Rauber T., *Parallel Programming: For Multicore and Cluster Systems*. New York: Springer-Verlag, 2010.
- [22] F. Alastair, P. Keir, and L. Anton, “Compile-time and run-time issues in an auto-parallelisation system for the Cell BE processor,” in *Proceedings of the 2nd Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC’08)*, vol. 5415 of *Lecture Notes in Computer Science*, pp. 163–173, Springer, 2008.
- [23] M. Hall and et la., “Improving the effectiveness of parallelizing compilers,” *ACM*,95, 1995.
- [24] N. DiPasquale, V. Gehlot, and W. T., “Comparative Survey of Approaches to Automatic Parallelization,” *MASPLAS’05*, 2005.
- [25] A. Ghuloum, “What makes Parallel Programming Hard,” 2007. <http://blogs.intel.com/research/2007/08/what makes parallel programmin.php>.
- [26] A. Guillon, “An apl compiler: The sofremi-agl compiler,” 1987. *ACM*,89.

Bibliography

- [27] H. Wei and J. Yu, "Mapping openmp to cell: An effective compiler framework for heterogeneous multi-core chip," *the 3rd international workshop on OpenMP*, 2008.
- [28] g. Russell, P. Keir, A. Donaldson, U. Dolinsky, A. Richards, and C. Riley, "Programming heterogeneous multicore systems using threading building blocks," in *Proceedings of the 2nd EuroPar Workshop on Highly Parallel Processing on a Chip (HPPC'08)*, HPPC 2010, 2010.
- [29] A. Donaldson, D. Dolinsky, A. Richards, and G. Russell, "Automatic offloading of c++ for the cell be processor: a case study using offload," *MuCoCoS10, IEEE Computer Society*, pp. 901–906, 2010.
- [30] A. Marowka, "Performance of OpenMP on Multicore Processors," in *ICA3PP: Proceedings of the 8th International Conference, 2008*, pp. 208–219, Springer-Verlag, Berlin Heidelberg, 2008.
- [31] D. Walker and J. Dongarra, "MPI: A Standard Message Passing Interface," *Supercomputer*, vol. 12, no. 1, pp. 56–68, 1996.
- [32] P. Jaaskelainen, C. Lama, P. Huerta, and J. Takala, "OpenCL-based design methodology for application-specific processors," *Embedded Computer Systems (SAMOS), 2010 International Conference*, pp. 223–230, 2010.
- [33] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language.," *In International Journal High Performance Comp.*, vol. 21, pp. 291–312, 2007.
- [34] G. Clemens and S. Scholz, "Sac: Off-the-shelf support for data-parallelism on multicores," pp. 25–33, 2007.
- [35] P. Cockshott and G. Michaelso, "Orthogonal Parallel Processing in Vector Pascal," *J. Comput. Lang. Syst. Struct.*, 32, pp. 2–41, 2006.
- [36] "Codeplay: Portable high-performance compilers," 2011. <http://www.codeplay.com/>.
- [37] "OpenHMPP, New HPC Open Standard for Many-Core," <http://www.openhmpp.org/en/OpenHMPPConsortium.aspx>, Online; accessed 14/9/2011.
- [38] A. Donaldson, C. Riley, A. Lokhmotov, and A. Cook, "Auto-parallelisation of sieve c++ programs," pp. 18–27, 2007.

Bibliography

- [39] R. Choy and A. Edelman, “Parallel MATLAB: Doing it Right,” *In International Journal High Performance Comp.*, 2003.
- [40] L. Snyder, *A Programmer’s Guide to ZPL*. Cambridge: MIT Press, 1999.
- [41] T. Turner, *Vector Pascal a Computer Programming Language for the Array Processor*. PhD thesis, Iowa State University, USA, 1987.
- [42] P. Cockshott, “Vector pascal reference manual,” *SIGPLAN*, vol. 37, no. 6, pp. 59–81, 2002.
- [43] J. David, J. Hall, and P. Trinder, “A strategic profiler for glasgow parallel haskell,” in *The Proceedings of the International Workshop on the Implementation of Functional Languages (IFL’98)*, (London), Sept. 1998.
- [44] S. Grelck and S. Scholz, “SAC- A Functional Array Language for Efficient Multi-threaded Execution,” *International Journal of Parallel Programming*, vol. 34, no. 4, 2006.
- [45] G. Blelloch, J. Hardwick, j. Sipelstein, and M. Zagha, “NESL user’s manual (for NESL version 3.1).,” *Technical Report CMU-CS-95-169*, 1995.
- [46] G. Blelloch, “Nesl: A nested data-parallel language,” vol. CMU-CS-95-170, Carnegie Mellon University, 1995.
- [47] M. Feldman, “Sun’s Fortress Language: Parallelism by Default,” *In International Journal High Performance Comp.*, 2008.
- [48] A. Noll, A. Gal, and F. M., “Cellvm: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor,” Tech. Rep. 06-17, Tech. Rep., 2006.
- [49] R. Mcilroy and J. Sventek, “Hera-JVM: Abstracting processor heterogeneity behind a virtual machine,” *The 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)*, 2009.
- [50] M. Harvey and G. Fabritiis, “Swan: A tool for porting CUDA programs to OpenCL,” *Computer Physics Communications*, vol. 182,no. 4, pp. 1093–1099, 2011.
- [51] “Cuda,” <http://wiki.mediacoderhq.com/index.php/CUDA>, 2009.
- [52] “Introduction to opencl programming,” (Cornell University), AMD, May, 2010.

Bibliography

- [53] R. Farber, “OpenCL, a Portable Parallelism,” tech. rep., The code project, 2010.
- [54] V. Srinivasa, A. Santhanam, and M. Srinivasan, “Cell broadband engine processor dma engines,” 2005. <http://www.ibm.com/developerworks/power/library/pa-celldmas/>.
- [55] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” *IBM journal of Research and Development*, vol. 49, no. 4, pp. 589–604, 2005.
- [56] A. Arevalo and et al., *Programming the Cell Broadband Engine Architecture*. International Technical Support Organization, 2008.
- [57] IBM, “Ibm cell training session notes,” (Baresbury Laboratory), April, 2009.
- [58] W. P. Cockshott, “Vector pascal an array language for multimedia code,” in *APL2002*, ACM SIGAPL, July 2002.
- [59] P. Shaw and G. Milne, “A highly parallel FPGA-based machine and its formal verification,” *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 162–162, 1993.
- [60] P. e. a. Cooper, “Offload automating code migration to heterogeneous multicore systems,” *HiPEAC, ser. LNCS*, pp. 337–352, 2010.
- [61] “Illiacy,” http://en.wikipedia.org/wiki/ILLIAC_IV, Online; accessed 12-August-2010.
- [62] P. Behrooz, *Introduction to Parallel Processing Algorithms and Architectures*. Kluwer Academic Publishers, 2002.
- [63] “The Cray X-MP Series of Computer Systmes,” 1985. <http://archive.computerhistory.org/resources/text/Cray/Cray.X-MP.1985.102646183.pdf>.
- [64] R. Russell, “The cray-1 computer system,” in *Computer Structures* (D. Sieworek, G. Bell, and A. Newell, eds.), McGraw Hill, 1985.
- [65] D. Hillis, *The connection machine*. Cambridge, MA: MIT Press, 1986.
- [66] “The Cray Y-MP Series of Computer Systmes,” 1985. <http://en.wikipedia.org/wiki/Cray-Y-MP>.
- [67] P. Ceruzzi, *A history of modern computing*. The MIT press, 2003.

Bibliography

- [68] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer* 41,7, pp. 33–38, 2008.
- [69] B. Schauer, "Multicore Processors - A Necessity," 2008. <http://www.csa.com/discoveryguides/multicore/review.pdf>.
- [70] A. Ross and J. Wood, "NVIDIA GeForce 256 DDR Guide ," 1999. <http://www.sharkyextreme.com/hardware/guides/nvidia-geforce256/6.shtml>.
- [71] "Nvidia cude, programming guide, version 2.1," tech. rep., Aug. 2008.
- [72] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pp. 18–23, IEEE Computer Society Washington, DC, USA, 2005.
- [73] J. Gorbod, "AMD Phenom II X6 1090T Black Edition," 2010. <http://www.bit-tech.net/hardware/cpus/2010/04/27/amd-phenom-ii-x6-1090t-black-edition/1>.
- [74] J. Abellan, J. Fernandez, and M. Acacio, "CellStats: a Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE," *Proceedings of 16th Euromicro International Conference on Parallel Distributed and network-based Processing*, pp. 261–268, 2008.
- [75] P. Homburg, M. Steen, and A. Tanenbaum, "Distributed shared objects as a communication paradigm," In *Proceedings of the 2nd Annual ASCI Conference*, pp. 132–137, 1996.
- [76] B. Bershad and M. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," 1991.
- [77] J. C. Rojas and M. Leeser, "Programming portable optimized multimedia applications," in *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, (New York, NY), pp. 291–294, ACM Press, 2003.
- [78] G. Grelck and S. Scholz, "Sac - from high-level programming with arrays to efficient parallel execution," *Parallel Processing Letters*, vol. 13, no. 3, pp. 401–412, 2003.
- [79] B. K. and ela, "Supporting OpenMP on Cell," *International Journal of Parallel Programming*, pp. 289–311, 2008.
- [80] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers* C-21, pp. 948–960, 1972.

Bibliography

- [81] C. Lomont, "Introduction to Intel Advanced Vector Extensions," *In Proceedings of the 2nd Annual ASCI Conference*, pp. 132–137, June 2011.
- [82] S. Reddaway, "Dap a distributed array processor," *SIGARCH Comput. Archit. News*, vol. 2, no. 4, pp. 61–65, 1973.
- [83] "P5 (microarchitecture)," 2011. [http://en.wikipedia.org/wiki/P5-\(micro_architecture\)](http://en.wikipedia.org/wiki/P5-(micro_architecture)), Online; accessed 11/8/2011.
- [84] AMD, "3dnow! technology manual," 2000.
- [85] K. Diefendorff, P. Dubey, R. Houchsprung, and H. Scales, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, 20, pp. 85–95, 2000.
- [86] Intel, "Picture the Future now Intel AVX," <http://software.intel.com/en-us/avx/>, 2011.
- [87] V. Konda and et al., "A simdizing c compiler for the mitsubishi electric neuro4 processor array," in *SUIF Workshop*, 1996.
- [88] "Data Structure Alignment," <http://en.wikipedia.org/wiki/Data-structure-alignment-RISC>, 2011.
- [89] L. Huang, L. Shen, Z. Wang, W. Shi, N. Xiao, and S. Ma, "Sif: Overcoming the limitations of simd devices via implicit permutation," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, 2010.
- [90] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Performance impact of misaligned accesses in SIMD extensions," *17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2006)*, pp. 334–342, 2006.
- [91] M. Scarpino, *Programming The CELL processor*. Prentice Hall, 2008.
- [92] B. Ramarkishna and J. Fisher, "Instruction-Level Parallel Processing:History, Overview and Perspective," tech. rep., Herlett Packard, Computer Systems Laboratory, 1992.
- [93] K. Asanovic and et la., "A view of the Parallel Computing Landscape," *Commun. ACM*, 52(10), pp. 56–67, 2009.
- [94] "Intel Threading Buidling Blocks," <http://en.wikipedia.org/wiki/Intel-Threading-Building-Blocks>.

Bibliography

- [95] “CUDA Data Parallel Primitives Library,” <http://code.google.com/p/cudpp/>, Online; accessed 11/8/2011.
- [96] M. Metcalf and J. Reid, *Fortran 90 explained*. Oxford University Press, Inc. New York, NY, USA, 1990.
- [97] A. Formella and et la., *The SPARK 2.0 system a Special Purpose Vector Processor with a VectorPASCAL Compiler*. The Twenty fifth Annual Hawii International Conference on System Sciences, HICSS 25, 1992.
- [98] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones, “GUM: a portable implementation of Haskell,” in *Proceedings of Programming Language Design and Implementation*, (Philadephia, USA), May 1996.
- [99] A. Giacalone, P. Mishra, and S. Prasad, “Facile: A symmetric integration of concurrent and functional programming,” *International Journal of Parallel Programming*, vol. 18, pp. 121–160, April 1989.
- [100] S. Scholz, “Sac- efficient support for high-level array operations in a functional setting,” *Journal of Functional Programming*, vol. 13, no. 6, pp. 1005–1059, 2003.
- [101] S. Scholz, “Functional array programming in sac,” Department of Computer Science University of Hertfordshire, 2005.
- [102] H. Yu and L. Rauchwerger, “Adaptive Reduction Parallelization Technique,” in *ICS’00: Proceedings of the 14th International conference on Supercomputing*, pp. 66–75, ACM New York, NY, USA, 2000.
- [103] K. Olukotun and L. Hammond, “A Future of Multiprocessors,” *ACM Transactions on Embedded Computing Systems*, , Publication date: April 2008, vol. 7, no. 3, pp. 26–29, 2005.
- [104] K. Iverson, *A programming language*. New York: Wiley, 1966.
- [105] IBM, “Language, time sharing system,” *IBM*. [Online; accessed 11/4/2011].
- [106] “APL (programming language),” [http://en.wikipedia.org/wiki/APL \(programming language\)](http://en.wikipedia.org/wiki/APL_(programming_language)), Online; accessed 7/12/2010.
- [107] K. Iverson, *Programming in J*. Iverson Software Inc, Toronto, 1992.
- [108] T. Budd, “An apl compiler for a vector processor,” *ACM Transactions on Programming Languages and Systems*, vol. 6, July 1984.

Bibliography

- [109] B. Hakami, *Efficient Implementation of APL in a Multilanguage Environment*. International Computers Ltd, 1975.
- [110] L. Bradford, “The design and implementation of a region-based parallel language,” in *PhD Thesis*, University of Washington, November 2001.
- [111] “ZPL (programming language),” [http://en.wikipedia.org/wiki/ZPL_\(programming_language\)](http://en.wikipedia.org/wiki/ZPL_(programming_language)), Online; accessed 4/4/2011.
- [112] S. Scholz, “Single assignment c (tutorial),” Department of Computer Science University of Hertfordshire, 2007.
- [113] C. Peter, “Porting the Vector Pascal Compiler to the Playstation 2,” Master’s thesis, University of Glasgow Dept of Computing Science, 2005.
- [114] J. Guo, J. Thiagalingam, and S.-B. Scholz, “Breaking the gpu programming barrier with the auto-parallelising sac compiler,” in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP ’11, (New York, NY, USA), pp. 15–24, ACM, 2011.
- [115] J. Backus, “The history of fortran i, li, and hi,” *IEEE Annals of the History of Computing*, 20(4), pp. 68–78, 1990.
- [116] “Fortran Language Standards,” <http://fortranwiki.org/fortran/show/Standards>, Online; accessed 7/10/2011.
- [117] M. Metcalf and J. Reid, *The F Programming language*. Oxford University Press Inc., New York, 1996.
- [118] “Fortran 77 standard,” <http://www.fortran.com/F77-std/rjcnf0001.html>, Online; accessed 13/9/2010.
- [119] “Fortran,” <http://en.wikipedia.org/wiki/Fortran-FORTRAN>, Online; accessed 16/9/2010.
- [120] R. Davies, A. Rea, and T. Dimitris, “Introduction to Fortran 90,” *An introduction Course for Novice Programmers*.
- [121] “Co-array fortran,” <http://www.co-array.org/>, Online; accessed 06/5/2012.
- [122] J. Smith and R. Nair, *Virtual Machines: Versatil Platforms for Systems and Processes*. ELSEVIEW Inc., 2005.
- [123] “Virtual Machine,” http://en.wikipedia.org/wiki/Virtual_machine, Online; accessed 13/10/2011.

Bibliography

- [124] “Clang Compiler User’s Manual,” 2009. <http://clang.llvm.org/docs/UsersManual.html>.
- [125] A. Dijkstra, J. Foker, and S. Swierstra, “The Architecture of the Utrecht Haskell Compiler,” *Haskell ’09, 2nd ACM SIGPLAN symposium on HaskellACM*, pp. 93–104, 2009.
- [126] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, and R. M. an V. Sunderam, *PVM: Parallel Virtual Machine A Users’ Guide and Tutorial for Network Parallel Computing*. 2003. To appear in Journal of Economic Behaviour and Organisation.
- [127] A. Ferrari, “Jpvm: Network parallel computing in java,” *ACM Workshop on Java for High Performance Network Computing*, 1998.
- [128] “Genetic algorithm,” [http://en.wikipedia.org/wiki/Genetic Algorithm Reproduction](http://en.wikipedia.org/wiki/Genetic_Algorithm_Reproduction), Online; accessed 21/2/2011.
- [129] W. Hosch, “Genetic algorithm,” <http://www.britannica.com/EBchecked/topic/752681/genetic-algorithm>, Online; accessed 11/05/2011.
- [130] B. Liu, R. Haftka, M. Akgun, and A. Todoroki, “Permutation genetic algorithm for stacking sequence design of composite laminates,” *Comput Meth Appl Mechan Eng*, vol. 186, pp. 357–372, 2000.
- [131] D. Patterson, “Reduced instruction set computers, risc,” vol. 28, No 1, Communication of ACM, 1985.
- [132] “TOP500 List - June 2011,” 2011. [<http://top500.org/list/2011/06/100>, Online; accessed 7/11/2011].
- [133] R. McIlroy, “Using program behaviour to exploit heterogeneous multi-core processors,” in *PhD Thesis*, Glasgow University, 2010.
- [134] “Ibm xl c/c++ for multicore acceleration for linux on x86 systems, v9.0 delivers cell broadband engine architecture application development capability,” 2007.
- [135] “Cell BE SDKs,” 2010. <http://www.bsc.es/projects/deepcomputing/linuxoncell/>, Online; accessed 19/12/2010.
- [136] A. e. l. Eichenberger, “Optimizing compiler for the cell processor,” in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’05, pp. 161–172, IEEE Computer Society, 2005.
- [137] T. Turner, *Vector Pascal a Computer Programming Language for the Array Processor*. PhD thesis, Iowa State University, USA, 1987.

Bibliography

- [138] A. Formella and et la., *The SPARK 2.0 system a Special Purpose Vector Processor with a VectorPASCAL Compiler*. The Twenty fifth Annual Hawii International Conference on System Sciences, HICSS 25, 1992.
- [139] P. Cockshott and G. Michaelso, "Orthogonal Parallel Processing in Vector Pascal," *J. Comput. Lang. Syst. Struct.*, 32,, pp. 2–41, 2006.
- [140] P. Cockshott, *Glasgow Pascal Compiler with Vector Extensions*. Glasgow University, 2011.
- [141] I. Jackson, "Opteron Support for Vector Pascal," Master's thesis, Dept Computing Science, University of Glasgow, 2004.
- [142] P. Cockshott, "Direct Compilation of High Level Languages for Multi-media Instruction Sets," 2000.
- [143] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori, "N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture," 2011.
- [144] "X86 assembly/x86 assemblers," <http://en.wikibooks.org/wiki/X86-Assembly/x86-Assemblers>, Online; accessed 22/8/2011.
- [145] "Assembler directives," <http://sources.redhat.com/binutils/docs-2.12/as.info/Pseudo-Ops.html-PseudoOnline>; accessed 19/10/2011.
- [146] "Lables," <http://sourceware.org/binutils/docs-2.17/as/Symbol-Names.html-Symbol-Names>.
- [147] C. Nelson, "Intel Core 17 "Nehalem" CPU Review," 2009. <http://www.hardcoreware.net/intel-core-i7-nehalem-cpu-review/>.
- [148] j. Breitbert and C. Fohry, "OpenCL, An Affective Programming Model for Data Parallel Computations at the Cell BE," *Parallel and Distribute Processing, Workshop and PhD Forum (IPDPSW)*, pp. 1–8, 2010.
- [149] P. Clarke, "Parallel Programming Tool Offered for Cell Processor," 2010.
- [150] N. Tsingos, "Using programmable graphics hardware for auralization," *In Proc. EAA Symposium on Auralization, Espoo, Finland*, 2009.
- [151] K. Group, "OpenCL Introduction and Overview," 2010. <http://www.khronos.org/developers/library/overview/opencl-overview.pdf>.

Bibliography

- [152] NVIDIA, *The CUDE Compiler Driver NVCC*. NVIDIA, January 2008.
- [153] Intel, “Intel threading building blocks,” Intel Corporation, <http://www.intel.com>, 2007.
- [154] Y. Gdura and P. Cockshott, “A Virtual SIMD Machine Approach for Abstracting Heterogeneous Multicore Processors,” *Journal of Computing (GSTF)*, vol. 1, pp. 143–148, January 2012.
- [155] M. Gschwind, “The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor,” tech. rep., IBM Research Division, 2006.
- [156] J. Abellan, J. Fernandez, and M. Acacio, “Characterizing the Basic Synchronization and Communication Operations in Dual Cell-Based Blades,” *Proceedings of 8th International Conference on Computation Science*, pp. 456–465, 2008.
- [157] A. Ershov, “On Programming of Arithmetic Operations,” pp. 3–6, 1958.
- [158] J. Flajolet, P. Amal and Raoult and V. J., “The number of registers required for evaluating arithmetic expressions,” *Theoretical Computer Science*, 9, pp. 99–125, 1979.
- [159] D. Warren and A. Center, “An abstract prolog instruction set,” tech. rep., SRI International, 1983.
- [160] A. Nisbet, “Genetic algorithm optimized parallelization,” *Workshop on Profile and Feedback Directed Compilation*, 1998.
- [161] N. Nwu and S. Li, “Instruction selection for arm thumb processors based on a genetic algorithm coupled with critical event tabu search,” *Chinese J. Computers*, Vol 40, no. 4, pp. 680–685, 2007.
- [162] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, “Genetic algorithms for the travelling salesman problem: A review of representations and operators,” *Artificial Intelligence Review* 13 (2), pp. 129–170, 1999.
- [163] D. Dasgupta and D. McGregor, “Nonstationary function optimization using the structured genetic algorithm,” *Parallel Problem Solving from Nature*, vol. 2, pp. 145–154, 1992.
- [164] A. Aho, W. Kernighan, and P. Weinberger, *The AWK Programming Language*. Addison-Wesley, 1988.
- [165] K. Sandeep, *Practical Computing on the Cell Broadband Engine*. Springer, 2009.

Bibliography

- [166] D. Playne, M. Johnson, and K. Hawick, “Benchmarking gpu devices with n-body simulations.,” International Conference on Computer Design(CDES 09), 2009.
- [167] “Challenge phaseii,” <http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge-PhaseII>, 2011.
- [168] “The Computer Language Benchmarks Game,” <http://shootout.alioth.debian.org>; accessed 11/11/2010.